Mejorando el acceso a la biblioteca de la UCM
Improving access to the UCM library

Trabajo de Fin de Grado
Curso 2019–2020

Autores
Miguel Ángel Castillo Moreno
Manuel María Guerrero Serrano
Mario Torres Cabañas

Directores
Alberto Díaz Esteban
Antonio Fernando García Sevilla

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Mejorando el acceso a la biblioteca de la UCM
Improving access to the UCM library

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Ingeniería del Software e Inteligencia Artificial

Autores
Miguel Ángel Castillo Moreno
Manuel María Guerrero Serrano
Mario Torres Cabañas

Directores
Alberto Díaz Esteban
Antonio Fernando García Sevilla

Convocatoria: *Junio 2020*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

26 de Junio de 2020

Dedicatoria

Quiero dedicar este proyecto a todos los amigos y familiares que me han apoyado. Son muchas personas como para ponerlas por nombre, pero sin cada una de ellas me hubiera sido imposible llegar hasta aquí. Gracias por todo.

- Miguel

A mi madre por ayudarme en todo lo que ha podido cuando más lo he necesitado. A mi padre por darme apoyo y ánimo en todo momento. A mi hermano por su sinceridad y su alegría contagiosa. A Julia por hacerme compañía cada tarde trabajando en este proyecto. Gracias.

- Manuel

Me gustaría dedicar este proyecto a mis padres, Carolina y José Ambrosio, y a mi pareja, Natalia, por apoyarme en los momentos difíciles y por darme confianza cuando no la tenía. Este proyecto es gracias a vosotros.

- Mario

Agradecimientos

A nuestros tutores, *Alberto Díaz Esteban* y *Antonio Fernando García Sevilla*, por el seguimiento constante, el apoyo a este proyecto, la ayuda proporcionada y, sobre todo, por su infinita paciencia. Gracias por todo.

Resumen

Mejorando el acceso a la biblioteca de la UCM

Este proyecto se trata de la continuación y mejora del trabajo realizado por otros estudiantes el año anterior. Con él, se pretende mejorar la funcionalidad del asistente de voz para la Biblioteca de la Universidad Complutense de Madrid llamado Janet, con la finalidad de que pueda ser utilizado por los estudiantes en un entorno real. La funcionalidad principal de esta aplicación es la búsqueda de libros dentro del catálogo de la UCM, aunque además de esto dispone de otras funcionalidades, como la consulta de horarios, direcciones y números de teléfono de las diferentes bibliotecas.

En este proyecto, se mejorará la capacidad de entender el diálogo por parte de Janet para poder dar la respuesta adecuada a cada situación, se estudiarán más plataformas desde las cuales poder acceder al servicio y se mejorarán tanto la accesibilidad como las funcionalidades del mismo.

Se pueden encontrar todos los recursos, así como ejecutables, archivos y código del proyecto en:

https://github.com/NILGroup/TFG-1920-Biblioteca

Palabras clave

Asistente virtual, Chatbot, Buscador, Biblioteca, Accesibilidad, Rasa, spaCY, OCLC, WMS

Abstract

Improving access to the UCM library

This project is a continuation and improvement of the work done by other students last year. Its scope is to improve the voice assistant for the library of Universidad Complutense de Madrid called Janet, aiming for it to be used in real environments. Its main functionality is the ability to search books inside the catalogue of the UCM, however, it also has other functionalities such as consult the schedules, locations, and phone numbers of the different libraries.

In this project, Janet's understanding of dialogue will be improved in order to give a proper answer, more platforms to support the service will be considered, and both accessibility and functionality will be enhanced.

All project's resources, executable files, code, and scripts, can be accessed from:

https://github.com/NILGroup/TFG-1920-Biblioteca

Keywords

Virtual assistant, Chatbot, Search engine, Library, Accessibility, Rasa, spaCY, OCLC, WMS

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

Vivimos en una sociedad en la que, si una aplicación no es fácil de utilizar, no existe. Cada cambio de un sistema operativo o herramienta de software trae siempre consigo mejoras que facilitan la vida al usuario. Antes, el simple hecho de utilizar un ordenador requería tener amplios conocimientos de informática, mientras que ahora, cualquier persona, sin importar su edad, es capaz de utilizar WhatsApp.

Ahora más que nunca, con la tecnología de los asistentes virtuales, cualquiera con pocas nociones de cómo manejar un dispositivo móvil puede acceder a funcionalidades más complejas. *Siri*, *Alexa* o *Ok Google* son palabras que escuchamos continuamente, y suponen una gran ayuda para un número cada vez mayor de gente.

En el caso de la biblioteca de la Universidad Complutense de Madrid, la cantidad de usuarios que utilizan los servicios online para acceder a ella no es muy elevado. Esto puede deberse a que no es lo suficientemente amigable, y a que los estudiantes desconocen las funcionalidades que se ofrecen.

Es por esto por lo que nuestros compañeros en el curso pasado se embarcaron en el proyecto de crear la aplicación conocida como *Janet*, la cual provee un chat de voz que nos permite hablar con una asistente virtual. Este asistente nos ayudará, entre otras cosas, a encontrar libros o incluso a conocer el horario de cada biblioteca o facultad.

Cautivados por *Janet*, no hemos podido evitar verle un futuro muy prometedor. No nos resulta difícil imaginar cómo los estudiantes podrían llegar a utilizarla de una forma habitual para cubrir sus necesidades, siempre y cuando ésta provea un servicio útil y efectivo.

1.2. Objetivos

Nuestra intención es ampliar y mejorar las funcionalidades ofrecidas por *Janet* para que sea una herramienta que pueda ser utilizada en un entorno real. A parte de mejorar su funcionamiento, los cambios estarán orientados a facilitar su desarrollo y mejora para poder llegar al punto en que la universidad lo pueda ofrecer como un servicio para todos los

estudiantes. Más importante aún, se quiere que a los estudiantes les resulte algo práctico y conveniente de utilizar.

Con este objetivo en mente, además de la aplicación de iOS y Android, queremos poder servir la aplicación a través de la web, para que pueda llegar al máximo número de personas y sea lo más cómoda posible.

1.3. Plan de trabajo

Para lograr la consecución de los objetivos establecidos, estudiaremos el funcionamiento del trabajo realizado por nuestros compañeros. Una vez la hayamos comprendido, observaremos qué áreas de la aplicación podemos mejorar y realizaremos cambios a estas. Además, crearemos una web y haremos que se pueda acceder a la aplicación desde ella, de manera similar a cómo se hace desde las aplicaciones móviles.

Nos organizaremos internamente utilizando Scrum para lograr lo planeado en el tiempo establecido, marcándonos metas pequeñas para lograr una evolución constante del proyecto. Así nos aseguraremos de no plantear objetivos inalcanzables, y poder realizar un trabajo tangible semana tras semana.

Chapter 2

Introduction

2.1. Motivation

We live in a society in which if an application is not easy to use, it does not exist. Each operative system's or application's evolution brings along quality of life improvements to its users. Previously, in order to use a computer you needed to have a good amount of knowledge about the subject, whilst right now, everyone, not minding its age, is able to use WhatsApp.

Now more than ever, with the technology of virtual assistants, anyone without any specialized knowledge about how to handle a mobile phone, can access more complex functionalities. *Siri*, *Alexa* or *Ok Google* are words we hear continuously, and they are helpful to an increasing number of people.

Regarding the UCM's library, it isn't very common to find people using its online services to access it. We believe that happens because it isn't friendly enough, and students aren't aware of the services that are being provided.

That is why our partners started with the project to create the application known as *Janet*. This app uses a voice chat that allows us to speak to a virtual assitant, who will help us, among other things, to find books and learn about the schedule of each library.

Captivated by *Janet*, we couldn't avoid thinking about its potential. We could imagine how students could end up using it on a daily basis to cover their needs, as long as the app provides a useful and effective service.

2.2. Objectives

Our intention is to extend and polish *Janet's* functionalities in order to make it a tool that can be used in a real environment. Apart from improving its performance, changes will be oriented to ease its development so it can get to the point where university can offer it as a service for all students. Moreover, we want to make it so that students will find it practical and convenient to use.

With these goals in mind, along the iOS and Android apps, we want to be able to serve it through the web, so we can reach the largest number of people possible and in order to be as comfortable as possible.

2.3. Workplan

In order to reach our objectives, we will study the app left by our seniors. After understanding it, we will analyze which areas need improving and we will make these improvements. Furthermore, we will create a web and we will make it so the app can be accessed through it, similar to how it is done from the mobile applications.

We will organize ourselves internally using Scrum in order to archive what we set up to do in the established time, setting ourselves small goals as to archive a constant evolution of the project. This way we will make sure we are not setting ourselves unreachable goals, and we will be able to make tangible work each week.

Capítulo 3

Estado del Arte

3.1. Servicios de la biblioteca de la UCM

La biblioteca de la Universidad Complutense de Madrid ofrece diversos servicios de forma física. Entre los más importantes, destacan:

- El préstamo de libros, documentos, e incluso de elementos de hardware como portátiles y ratones.
- La reserva de salas de trabajo en grupo.
- Servicios de información sobre la biblioteca y consulta en sala.
- Programas de apoyo a la docencia y a la investigación.

Aunque estos servicios son muy relevantes para el desarrollo educativo, nos vamos a centrar en los servicios electrónicos que ésta ofrece. Entre ellos, el más importante es el acceso a los recursos de información electrónicos como bases de datos, revistas o libros electrónicos o portales científicos. Esto permite disponer del conjunto de información de la Complutense desde cualquier parte, mientras se tenga conexión a internet.

Para el acceso a estos recursos se hace uso del **catálogo Cisne**. Este es un servicio que de forma online permite el acceso a libros, revistas, documentos, tesis almacenadas en la complutense, la producción científica de la universidad y diversos materiales electrónicos. Este servicio está basado en **WorldCat Discovery**, el mayor catálogo colectivo de bibliotecas del mundo, creado por OCLC.

OCLC es un cooperativa universitaria mundial, que recopila información útil alrededor del mundo. OCLC ofrece también varios servicios para cubrir las necesidades que puedan surgir en las bibliotecas universitarias. WorldCat Discovery es uno de estos servicios, la base de datos con todo el conocimiento colectivo, sostenida por todos sus socios.

3.1.1. Tecnologías de acceso a la información

El WorldCat Discovery dispone de múltiples APIs[1] con las que acceder a varios de los servicios ofrecidos. Las más relevantes son:

[1] (`https://platform.worldcat.org/api-explorer/apis`)

- **WorldCat Discovery API:** ofrece la posibilidad de buscar recursos en WorldCat. Está en fase beta, por lo que solo se proporciona el acceso a algunas de las universidades participantes. Afortunadamente, la UCM es una de ellas.
- **WMS Availability API:** otorga información sobre la disponibilidad de recursos específicos en una librería particular.
- **WMS Search API:** permite el acceso de bajo nivel a las librerías y las localizaciones.

Estas tres son APIs que se utilizan para obtener información íntegra sobre los libros y recursos electrónicos disponibles. Combinando las 3 se pueden buscar libros en la biblioteca de la UCM, obtener las bibliotecas en las que se encuentran, y comprobar el número de ejemplares disponibles de cada recurso.

Sin embargo, ninguna de estas APIs proporciona de forma directa la portada de los libros, por lo que para esto existe otra diferente. La API de Open Library Covers[2] permite obtener la imagen de portada de un libro dado su ISBN o código OCLC. Ambos se pueden obtener con la API de WMS Search. Utilizando este conjunto de herramientas, se puede obtener una visión muy completa sobre un libro que se desee buscar.

El inconveniente de estas APIs es que no pueden ser utilizadas por cualquiera. Sólo las bibliotecas participantes en el sistema de OCLC pueden obtener acceso a ellas. Estas claves fueron obtenidas de la UCM por nuestros compañeros el año pasado, y una implementación de las búsquedas de libros y portadas fue realizada por ellos.

3.2. Chatbots

Con la evolución de los sistemas informáticos, se pretende intentar que estos lleguen cada vez a más personas. En el comienzo de la informática, la forma de interactuar con estos sistemas era excesivamente compleja; solo unos pocos usuarios avanzados sabían qué hacer para obtener resultados. Poco a poco se ha ido simplificando la interacción con los equipos, abriendo la posibilidad de acceder a estos sistemas a usuarios que no estén especializados.

Incluso con todos los avances alcanzados, seguimos dependiendo de varios elementos de hardware como pueden ser un teclado o una pantalla táctil para interactuar con un ordenador. Aunque se den por supuestas estas formas de intercambiar información, son métodos que el ser humano no utilizaría de forma natural. No es sino un paso evidente el hecho de que un ser humano quiera comunicarse con un ordenador de la misma manera que se comunicaría con otra persona. La forma más común en la que dos personas suelen interactuar son **la voz y el lenguaje**.

Es por esto por lo que surge la idea de los chatbots. "Un **chatbot** es un sistema software, que puede interactuar o "chatear" con un usuario humano en un lenguaje natural como puede ser el inglés" (Shawar y Atwell, 2007). Ya no se utiliza un conjunto delimitado de comandos que el ordenador sepa interpretar, sino que se permite al usuario utilizar toda la extensión del vocabulario de su lenguaje para comunicarse con él.

[2] (`https://openlibrary.org/dev/docs/api/covers`)

Esta nueva forma de comunicarse con un sistema informático abre muchas posibilidades. Ahora, una persona que nunca ha interactuado con un ordenador puede comunicarse con él mediante la voz. Además, con el paso del tiempo, estos bots están aumentando tanto en rendimiento como en complejidad. Ahora un chatbot puede realizar funciones que serían impensables en un pasado.

3.2.1. Uso comercial de los chatbots

Por supuesto, el uso de los chatbots se ha ido extendiendo a lo largo de los años al ámbito comercial. Cada vez más empresas deciden implementar algún tipo de chatbot dentro de sus sistemas. Gracias a su increible versatilidad, es fácil encontrarle un lugar a estos bots dentro de una compañía.

Uno de los usos más ansiados por empresas suele ser como mecanismo para resolver **preguntas frecuentes**. Es muy llamativo tener una forma de resolver rápidamente las dudas de los clientes, además de ser interacción muy natural y humana. A menudo, estos chatbots se enmascaran bajo la premisa de que se está hablando con un ser humano real. Cuando ese chatbot no sabe responder algo, también se puede poner en contacto al cliente con el servicio técnico real, solucionando los fallos que pueda tener. Uno de los principales beneficios de estos bots es que son sencillos de implementar, ya que no tienen funcionalidades excesivamente complejas y trabajan sobre un dominio muy delimitado.

Otra forma en la que se pueden utilizar es **automatizando servicios de reservas o compras**. Por ejemplo, la aerolínea KLM (Villar, 2018) ofrece la opción de utilizar un chatbot para gestionar toda la información de reserva del vuelo mediante el chat de Facebook Messenger, como observamos en el ejemplo de la Figura 3.1. El sector del turismo es uno de los que más está implementando bots para gestionar sus servicios.

Una variante de chatbot que se utiliza muy a menudo son los **asistentes virtuales** como *Alexa* o *Siri*. Este tipo de asistentes se incorporan en todos los teléfonos modernos. Estos asistentes están diseñados principalmente para ayudar con funciones útiles en la vida cotidiana y simplificar el uso del dispositivo para usuarios menos expertos. La función de un asistente virtual va un poco más allá de la de un chatbot, realizando acciones que van más allá de una simple respuesta. Sin embargo, el principio que hay detrás es el mismo: tener que entender el lenguaje del usuario para generar una respuesta adecuada a su petición.

Estos casos de uso de chatbots son tan solo ejemplos de la versatilidad de los mismos. Es por esto por lo que son tan reclamados en la actualidad, y se han convertido en todo un éxito. Su popularidad va en continuo aumento, y cada vez un mayor número de empresas implementa un chatbot en alguna de sus formas.

3.2.2. Funcionamiento de los chatbos

El rendimiento y la complejidad de los chatbots se ha incrementado con la mejora de las tecnologías utilizadas para implementarlos. Desde los más básicos hasta los más complejos, estos pueden ser separados en dos tipos claramente diferenciados según las bases de su funcionamiento. Estos dos tipos son los **basados en reglas**, y los **basados en IA**.

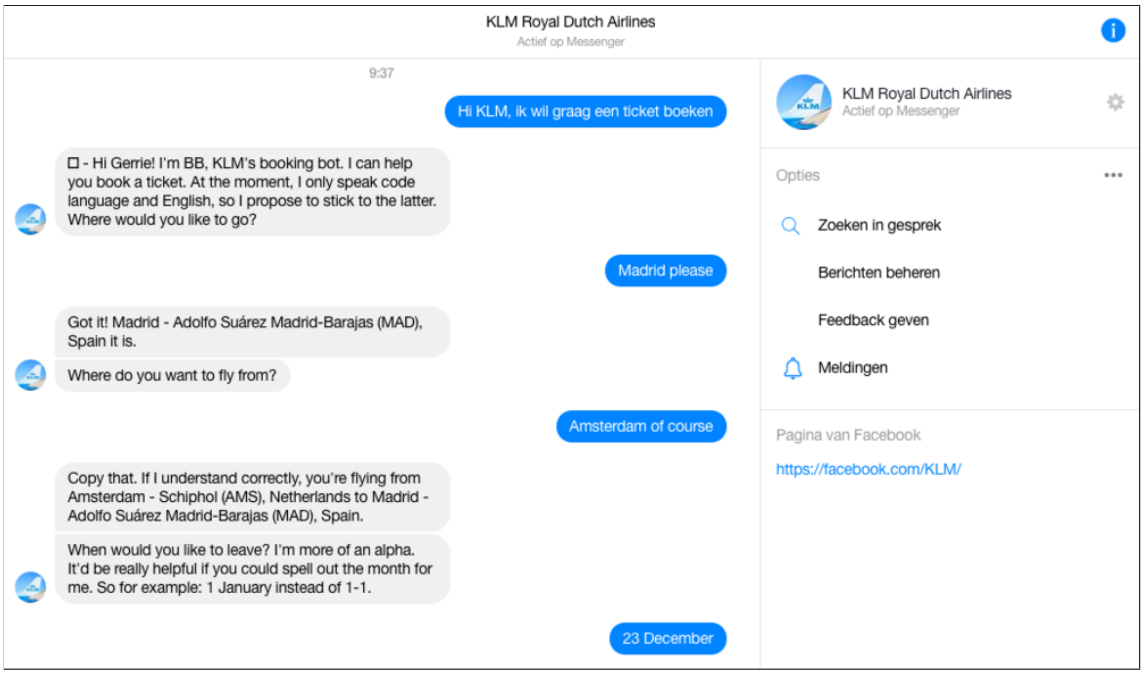

Figura 3.1: Ejemplo de utilización del chatbot proporcionado por la aerolínea KLM

3.2.2.1. Chatbots basados en reglas

El primer chatbot fue denominado **Eliza** (Bradesko y Mladenic, 2012). Este era un bot sencillo cuyo objetivo era actuar como un psicológo. Su forma de actuar era intentando relacionar patrones en las palabras que se introducían y dar la respuesta programada para dicho patrón. No era muy efectivo, pero al principio varios usuarios comentaron que daba la impresión de estar hablando con un ser humano.

Eliza es un ejemplo del tipo más primitivo de chatbot: el basado en reglas. Un chatbot basado en reglas o "rule-based chatbot" es aquel que responde preguntas en función de reglas establecidas por el programador. Estas reglas pueden ser desde muy simples a muy complejas. Un ejemplo de regla es, si el usuario dice de manera exacta "hola", el bot le debe responde "Hola, ¿qué tal?".

Un bot de este tipo es determinista. Para una misma entrada siempre va a responder con la misma salida. Sin embargo, esto no significa que sea completamente rígido. Este tipo de bot puede también extraer algunas palabras de lo que se le comunique, como el nombre del usuario, y utilizarlo para elaborar respuestas parametrizadas.

El principal inconveniente de este tipo de bot es que es incapaz de comprender el contexto en el que se encuentra. Por ello, le será difícil mantener una conversación de la misma forma en la que lo haría un ser humano. Para solucionar este problema, surgen los chatbots basados en IA.

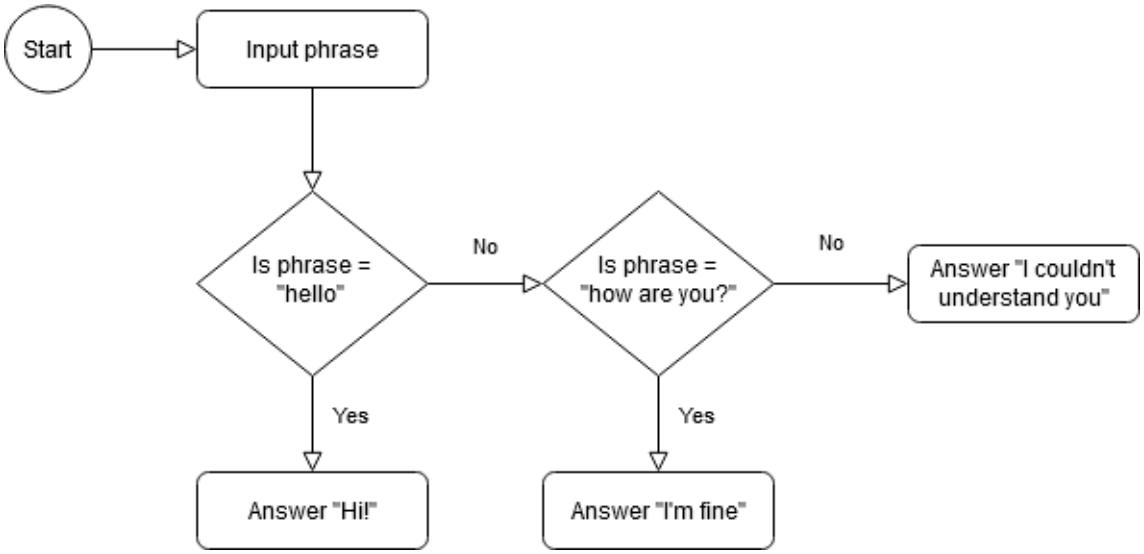

Figura 3.2: Ejemplo simple de una regla para un chatbot basado en reglas

3.2.2.2. Chatbots basados en inteligencia artificial

Una conversación con un ser humano es mucho más compleja de lo que pudiese parecer a simple vista. Una frase como "Sí, dámela", solo tiene sentido si tenemos en cuenta el contexto que la precede. Solo si sabemos que antes de esa frase otra persona le ha preguntado "¿Quieres que te de la comida?", la primera frase cobra sentido. Con este sencillo ejemplo, podemos observar varios aspectos importantes sobre el lenguaje:

- **El contexto:** Saber de qué se está hablando permite dar sentido a frases que por sí mismas pueden significar múltiples cosas. En el ejemplo dado, sabemos que la persona que responde quiere la comida sólo porque antes se le ha preguntado sobre ello. Solo relacionando varios aspectos de la conversación se pueden otorgar significados a cada parte de la misma.

  No solo es importante el contexto de la frase anterior. El contexto también puede ser el lugar en el que se realiza la conversación, con quién se realiza, el momento en el tiempo, o incluso se puede transmitir mediante comunicación no verbal. Es este casi infinito número de posibles variables el que hace tan complejo entender el significado una oración.

- **La intención:** En una conversación, las palabras siempre se dicen con un propósito más o menos claro. En el ejemplo, la intención de decir "Sí, dámela", es tanto afirmar la pregunta como pedir la comida.

  Se puede observar con esto que una persona siempre va a tener un objetivo en mente cuando enuncia una frase, y va a esperar un determinado tipo de respuesta. Es muy relevante para un chatbot poder entender qué es lo que pretende conseguir el usuario con una acción, y poder tomar la acción correspondiente y adecuada. Sin embargo, no es trivial determinar exactamente qué pretende decir un usuario, y este también puede tener múltiples intenciones con una sola oración.

Para resolver estos problemas complejos surgen los bots basados en inteligencia artificial. Estos son capaces, en mayor o menor medida, de entender tanto el contexto en el que se está hablando como la intención del usuario. Estos bots se basan en el procesamiento

Ventajas	Desventajas
▪ Implementación sencilla ▪ Rapidez tanto en creación como utilización ▪ Sencillo de ampliar en funcionalidad	▪ Respuestas estáticas ▪ No puede comprender el contexto ▪ Dificultades con preguntas no establecidas en las reglas

Tabla 3.1: Ventajas y desventajas de los bots basados en reglas

del lenguaje natural para realizar estas tareas.

"El procesamiento del lenguaje natural (PLN, o en inglés, NLP) se refiere a los sistemas informáticos que analizan, intentan entender, o producir uno o más lenguajes humanos, como el inglés, japonés, italiano o ruso" (Allen, 2003). La dificultad de entender el lenguaje se basa en varios aspectos ambiguos del mismo, como por ejemplo la ambigüedad léxica (banco de peces, o banco para sentarse), o la ambigüedad sintáctica ("Le vi con un telescopio" puede interpretarse como que se utilizó un telescopio para verle, o que tenía un telescopio cuando fue observado). Los sistemas de PLN utilizan diversos métodos de interpretación semántica y sintáctica para traducir una oración a un lenguaje que pueda ser comprendido por una base de datos. Es por esto por lo que los sistemas que mejor funcionan son aquellos con una base de datos acotada a cierto dominio; es mucho más intrincado trabajar sobre todo el lenguaje que sobre una parte del mismo.

Como subtópico del NLP surge **el entendimiento del lenguaje natural (ELN, o más conocido por sus siglas en inglés, NLU)**. Su propósito se basa en la comprensión lectora: entender el significado verdadero de un texto. Dentro de las tareas en las que se utiliza el NLU están la clasificación textos, la extracción de intenciones, y el análisis a gran escala del contexto.

Es gracias a la inteligencia artificial del NLU que se puede comprender la intención del texto del usuario dentro de un contexto, y por la que los chatbots basados en IA son capaces de comprender mucho mejor al usuario si se utiliza de manera correcta.

3.2.3. Comparación entre tipos de chatbots

Tras haber discutido los tipos de chatbots y cómo se comportan, se observar las ventajas y ventajas y desventajas de los basados en reglas (Tabla 3.1) y los basados en IA (Tabla 3.1).

Como se puede observar, los chatbots basados en IA ofrecen ventajas en funcionalidad tan solo a costa de complejidad y tiempo. Mientras que muchas empresas elegirán la opción de las reglas por el ahorro en tiempo y costes que pueda suponer, un bot siempre será más competente si es implementado utilizando mecanismo de inteligencia artificial.

Ventajas	Desventajas
▪ Compresión del contexto ▪ Evolución y aprendizaje a lo largo de su ciclo de vida ▪ Entendimiento de las intenciones ▪ Interacción más compleja y humana con el usuario	▪ Mayor complejidad ▪ Mayor tiempo tanto de implementación como de entrenamiento y respuesta

Tabla 3.2: Ventajas y deventajas de los bots basados en IA

3.3. Aplicación Janet

La aplicación llamada *Janet* (Loureiro y Varillas, 2019) es un asistente virtual con la capacidad de acceder a los servicios informáticos de la biblioteca de la UCM. Fue implementado como un chatbot de inteligencia artificial por alumnos de la UCM en el año 2019, utilizando como base el framework de Rasa (3.4).

Junto a *Janet*, se creó una aplicación para móvil nativa de iOS y otra para Android, las cuales proporcionan el acceso por voz a las utilidades de la herramienta. Entre las funcionalidades que ofrece están:

- Búsqueda de libros en la biblioteca, tanto por título, autor, tema, o una combinación de estos tres.
- Obtención de los horarios de apertura y cierre de las diversas bibliotecas.
- Obtención del número de teléfono de las bibliotecas.
- Obtención de la localización de las bibliotecas y facultades de la universidad.

Además, puede mantener conversaciones simples que permiten comúnmente los chats con bots. Es capaz por ejemplo de responder a saludos, despedidas o insultos. Sin embargo, no es capaz de responder sobre cualquier tema. Al fin y al cabo, *Janet* no es un bot de propósito general. A pesar de ello, es capaz de trabajar con las consultas que un usuario pueda hacer sobre algo ajeno al dominio del bot: cuando no se reconoce la intención en una consulta o se reconoce que está hablando de algún tema desconocido, se responde con una respuesta genérica, que informa al usuario sobre para qué y para qué no está preparada y de esta manera reconducir al usuario hacia preguntas que *Janet* sí sepa responder.

3.3.1. Arquitectura de la herramienta

En el momento de continuar con el desarrollo de *Janet*, la herramienta disponía de los siguientes módulos, conectados como se muestra en la Figura 3.5:

- Aplicación nativa para iPhone e iPad (Figura 3.3)[3] compatible con dispositivos con iOS 10.3.3 o posterior, desarrollada en Swift 4.2.

[3](`https://apps.apple.com/us/app/janet/id1451052771?l=es#?platform=iphone`)

Figura 3.3: Capturas de pantalla de *Janet* iOS

- Aplicación nativa para dispositivos con Android Lollipop o superior (Figura 3.4)[4] desarrollada en Java 1.8.0_191.
- Un módulo *back end* ocupado de traducir los datos entre clientes y controlador escrito en Python 3.6.
- Un controlador escrito en Python 3.6, intermediario entre los distintos módulos.
- Un módulo PLN encargado de la interpretación de los mensajes del usuario y en la decisión de la respuesta mediante el uso de Rasa 0.13.0[5] y SpaCy.
- Un módulo *buscador biblioteca* encargado de hacer consultas al catálogo de la biblioteca mediante el uso de los servidores de OCLC.
- Un módulo *DAO* que se comunica con una base de datos en MongoDB en la que se almacena información relevante de las bibliotecas e interacciones que realizan los usuarios.

3.3.2. Entorno de ejecución

Todos los módulos mencionados anteriormente, salvo las aplicaciones móviles, debían ser ejecutados de manera remota en un servidor. Para ello, también se proporcionan instaladores funcionales para Debian Stretch[6] y Ubuntu Bionic[7]. En concreto se facilitan tres

[4] (`https://play.google.com/store/apps/details?id=ucm.fdi.android.speechtotext&hl=es_419`)
[5] (`https://legacy-docs.rasa.com/docs/nlu/0.13.0/`)
[6] (`https://www.debian.org/releases/stretch/`)
[7] (`https://releases.ubuntu.com/18.04.4/`)

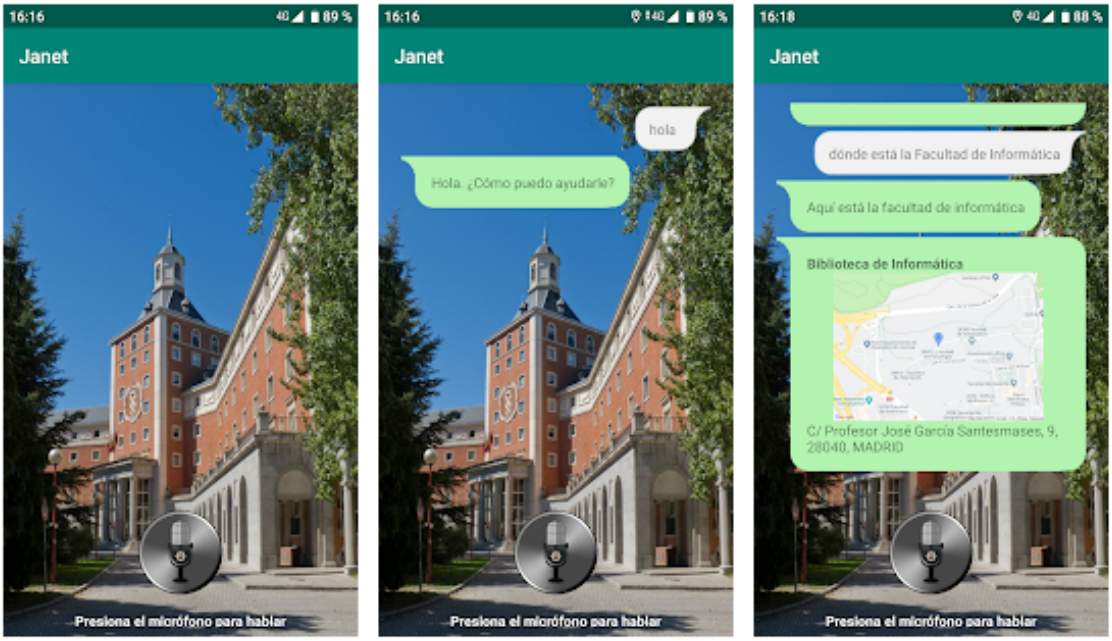

Figura 3.4: Capturas de pantalla de Janet Android

instaladores diferentes: uno para instalar la parte referente al módulo de procesamiento de lenguaje natural, otro para instalar lo demás y un tercero que engloba ambos y dispone de todo lo necesario para que *Janet* sea funcional. Los tres instaladores requieren ser ejecutados con permisos de superusuario y comprueban la existencia de las carpetas conteniendo sus correspondientes funcionalidades. Además, se necesita la existencia del archivo *wskey.conf*, que contiene información de acceso a las APIs de WorldCat (3.1.1). Finalmente, tras terminar la instalación, el sistema dispone de 3 servicios ejecutados mediante systemctl[8], cada uno ocupado de gestionar cierta funcionalidad de Janet:

- jarvis.service, ocupado de la gestión del procesamiento de lenguaje natural.
- jarvisactions.service, ocupado de ejecutar acciones complejas que pueda realizar *Janet*, como buscar en los servidores OCLC o en la base de datos.
- janet.service, que se encarga de los módulos back end y actúa como controlador.

3.4. Rasa

Rasa Open Source (Rasa, 2020) es el framework de machine learning en el que se fundamenta el módulo de procesamiento de lenguaje de *Janet*. Está escrito en Python y se usa para construir asistentes contextuales basados en conversaciones de voz y texto. Éste incluye:

- **Rasa NLU:** una herramienta de procesamiento de lenguaje natural que determina la información contextual más importante de un mensaje y lo que el usuario espera tras enviarlo. Funciona mediante la clasificación de intenciones y la extracción de entidades.

[8] (`https://www.freedesktop.org/software/systemd/man/systemctl.html`)

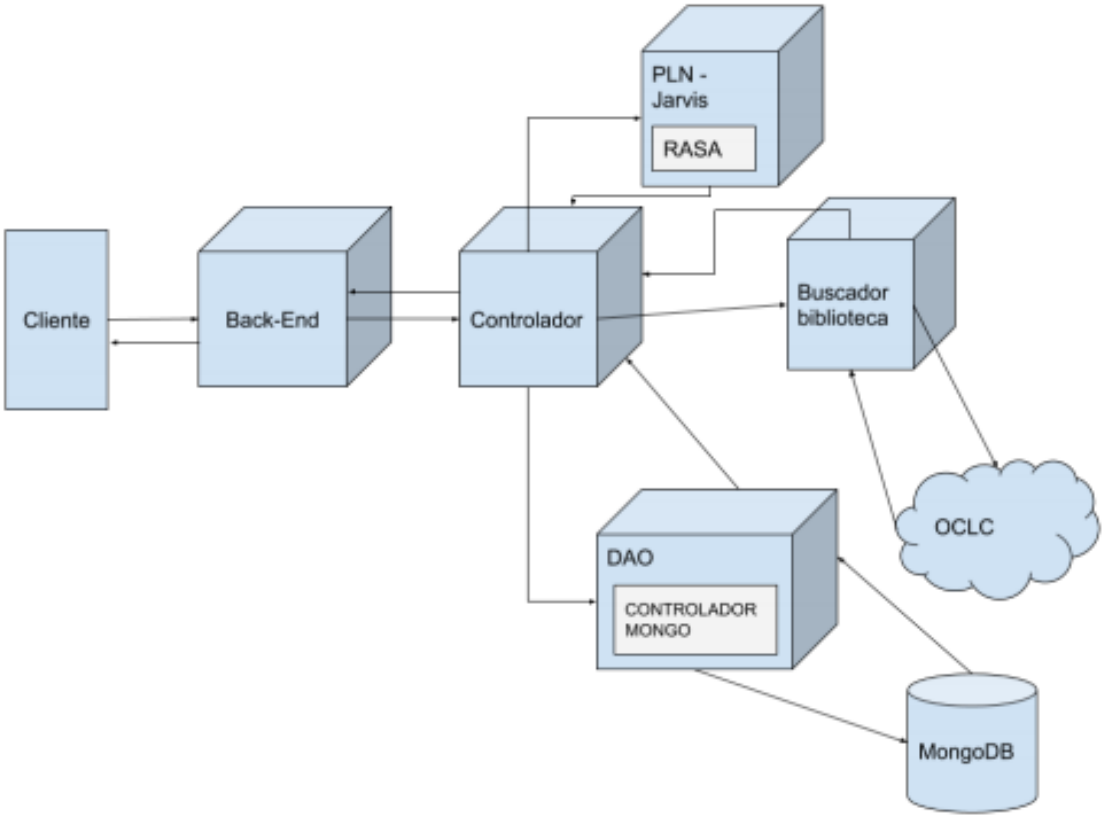

Figura 3.5: Representación de los módulos proporcionada en el primer proyecto de *Janet*

- **Rasa Core:** un motor de diálogo para construir asistentes de inteligencia artificial. Decide la mejor acción o respuesta basándose en el historial de la conversación.
- Un conjunto de herramientas que facilitan la conexión del asistente con los usuarios y los sistemas de back end.

Además, existe Rasa X, una herramienta que, mediante una interfaz gráfica, facilita la mejora de asistentes ya construidos con Rasa Open Source.

3.4.1. Rasa NLU

Como ha sido mencionado previamente, Rasa NLU es una herramienta de procesamiento de lenguaje natural, esto es, un mecanismo para obtener y procesar la información más relevante de un mensaje introducido en lenguaje humano.

Ejemplos de entrenamiento, intenciones y entidades

En Rasa NLU, la forma de reconocer dicha información es mediante el uso de ejemplos de entrenamiento, distinguiendo en ellos intenciones y entidades. Una intención, o intent, es, como su nombre indica, la acción que debería asociarse a un mensaje obtenido del usuario, mientras que una entidad, o entity, es el objeto al que se le dan valores específicos

sobre los cuales se pretende realizar dicha acción. Los datos de entrenamiento se pueden escribir usando Markdown o JSON y pueden ser dispuestos en un único archivo o en una carpeta conteniendo todos los ficheros necesarios. Los ejemplos son agrupados por sus correspondientes intenciones, y para cada ejemplo, si es necesario, se delimitan las palabras o grupos de palabras que dan un valor específico a una entidad, siendo acompañadas del nombre de dicha entidad. Por ejemplo, en la Figura 3.6 se muestra un ejemplo de entrenamiento: aquí, se está diciendo que los mensajes con esa forma tienen la intención de averiguar una ubicación y la entidad esperada sería una localización, que en este caso tendría el valor específico de la Biblioteca de Odontología.

```
## intent:consulta_localizacion
- busca la ubicación de la [Biblioteca de Odontología](localizacion)
```

Figura 3.6: Ejemplo de entrenamiento para el motor NLU

Además, para mejorar el nivel de confianza a la hora de reconocer intenciones, se pueden usar:

- Sinónimos: muestran maneras distintas de llamar a un mismo valor de una entidad.
- Expresiones regulares: ayudan a reconocer valores de entidades que van a tener una estructura determinista como puedan ser códigos postales o correos electrónicos.
- Tablas de búsqueda: listas de valores específicos que puede tomar cada entidad.

Pipeline de NLU

Una vez definidos los ejemplos de entrenamiento, hay que elegir una pipeline de NLU, que define los pasos por los que tiene que pasar un mensaje del usuario para poder reconocer su intención. Rasa NLU proporciona pipelines de NLU prediseñadas, tanto con conocimiento del lenguaje, útil para los casos en que se tienen pocos ejemplos de entrenamiento, como sin conocimiento del significado de las palabras, lo que ayuda a la personalización de la connotación de las mismas. Por otra parte, se puede utilizar una pipeline personalizada, creada listando los componentes que queremos utilizar, describiendo así los pasos por los que pasará el mensaje. Entre ellos se pueden encontrar:

- **Fuentes de vectores palabra**: proporcionan, para un idioma dado, una representación del lenguaje en la que las palabras son vistas como vectores, de tal manera que si dos palabras tienen significados similares, sus vectores apuntarán en direcciones similares, como se puede ver en la Figura 3.7. Rasa proporciona fuentes ya entrenadas como pueden ser las de *MITIE*, *spaCy* o *ConveRT*, además de permitir utilizar fuentes personalizadas con las que se puede enfatizar los significados que se quiere que tenga las palabras.
- **Tokenizadores**: separan los mensajes en tokens, normalmente palabras, eliminando los espacios y los símbolos de puntuación.
- **Extractores de características (featurizers)**: para entrenar un modelo de machine learning necesitamos datos numéricos. Es por ello que es necesario traducir las frases en este tipo de datos. Normalmente se usan vectores binarios en los que se determina si la palabra aparece en la frase o no.

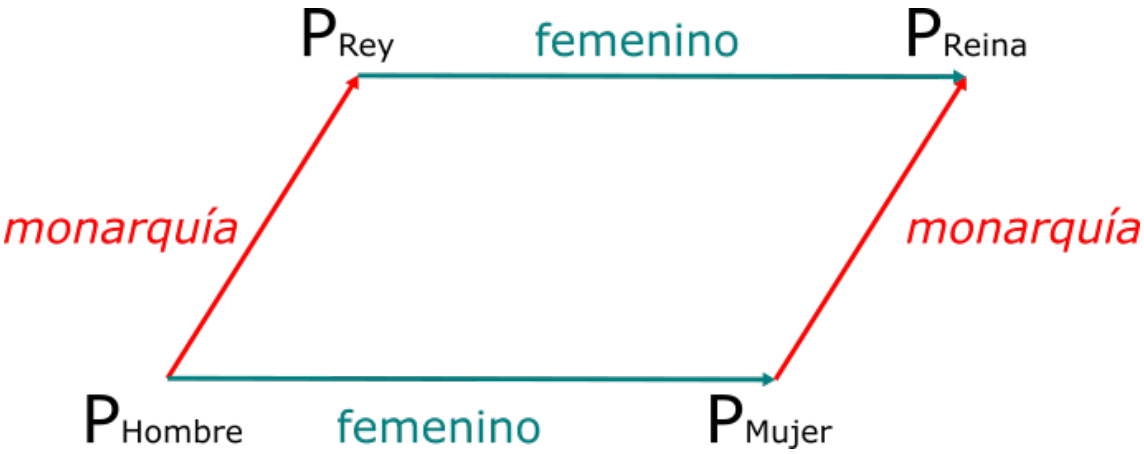

Figura 3.7: Ejemplo de representación de palabras como vectores

- **Clasificadores de intenciones** (Wochinger, 2019): modelos de machine learning de aprendizaje supervisado que aprenden a clasificar los mensajes por sus correspondientes intenciones.

- **Extractores de entidades**: diferentes métodos para adjudicar valores específicos a las entidades o crear relación entre dichos valores y sus sinónimos. Rasa permite el uso de distintas tecnologías para esto, desde campos aleatorios condicionales hasta transformers, pasando por máquinas de soporte vectorial o gramáticas libres de contexto.

3.4.2. Rasa Core

Rasa Core es un motor de diálogo que utiliza un modelo de machine learning entrenado en conversaciones de ejemplo para decidir qué hacer dependiendo del mensaje y del contexto.

Historias

La forma que tiene Rasa Core de entrenar el modelo de administración de diálogo es mediante las historias. Una historia es una representación de una conversación entre un usuario y el asistente, donde la entrada, por parte del usuario, son las intenciones y entidades reconocidas, y la respuesta del asistente se expresa con los nombres de las correspondientes acciones. El formato de las historias consta de un título que suele ser usado para describir la historia y una sucesión de los mensajes de los usuarios formateados a modo de intención con la respectiva entidad y valor, si aplican, junto a la acción o acciones que debe realizar el asistente ante dicha intención del usuario en dicho contexto.

Además, Rasa permite la opción de activar el aprendizaje interactivo, mediante el cual se le da feedback al bot mientras se habla con él. El funcionamiento es el siguiente: para cada input del usuario, se predice una intención junto a su nivel de confianza, y se pregunta si se ha acertado; si acierta sigue el diálogo, pero en caso contrario, se selecciona cual era la intención correcta. Con esto se generan nuevas historias con las que se podrá volver a entrenar al modelo, aumentando su precisión a la hora de decidir.

Como ejemplo, en la Figura 3.8 se muestra una historia generada automáticamente por Rasa mediante interacción con el asistente, de ahí el nombre no descriptivo. En ella se puede ver como se detecta una intencion de consultar un libro dado su título, cuyo valor aparece junto a la entidad *libro*. Tras esto, aparecen las acciones a ejecutar, que será realizar una búsqueda del libro usando sólo su título.

```
## Generated Story -2292381114852692301
* consulta_libros_titulo{"libro": "Narnia"}
    - form_libros
    - form{"name": "form_libros"}
    - slot{"libro": "Narnia"}
    - slot{"autores": null}
    - form{"name": null}
    - slot{"requested_slot": null}
```

Figura 3.8: Ejemplo de historia de entrenamiento

Finalmente destacar que las historias pueden ser modularizadas gracias al uso de checkpoints, mediante los cuales se puede generalizar partes de historias que se repiten, además de con el uso de *OR*, que permite extender historias que comienzan de la misma manera.

Acciones y eventos

Rasa Core tiene dos tipos de acciones distinguidas: las acciones de expresión (utterance actions), mensajes escritos con los que el asistente puede responder, y acciones personalizadas, con las que se ejecuta un código definido para dicha acción.

Además, acompañando a las acciones, se pueden incluir eventos que ayudan a guardar u ordenar información dada por los usuarios. Por ejemplo, los eventos de slot son usados para, tras realizar una acción, mantener un valor que podrá ser usado posteriormente, los eventos de formulario se utilizan para definir comportamientos que requieren varios pasos por parte de los usuarios, o los recordatorios provocan que cierta acción sea ejecutada en un tiempo especificado.

El Dominio

Para el correcto funcionamiento de Rasa Core, se define un Dominio que determina el entorno en el que funciona el asistente. En él, se especifican las intenciones, las entities y las acciones con las cuales el bot va a interactuar. También contiene el texto correspondiente a las acciones de expresión. Cabe remarcar la existencia de las sesiones, que permiten determinar cuando ha acabado una conversación con un usuario basándose en el

tiempo transcurrido o en determinadas acciones, así como para cambiar el contexto de la conversación.

Políticas

Otro elemento crucial para la decisión de la deriva de la conversación son las políticas, es decir, las reglas en las que se basa el agente para decidir qué acción tomar en cada paso de la conversación. Entre ellas se pueden encontrar:

- Keras Policy: Utiliza una red neuronal basada en LSTM para tomar la siguiente decisión.
- TED (Transformer Embedding Dialogue) Policy: Tiene una arquitectura predefinida de tipo transformer para la cual se concatenan las intenciones y entidades del usuario y las acciones previas del sistema en un vector con el que se predice la acción de salida y se compara con las acciones del sistema, eligiendo la más similar.
- Mapping Policy: Mapear ciertas intenciones directamente con unas acciones. Una intención sólo se puede mapear a, como máximo, una acción.
- Memorization Policy: Memoriza las conversaciones en los datos de entrenamiento. Si una conversación coincide con una de entrenamiento, predice la siguiente acción con un 100 % de confianza, y con un 0 % en caso contrario.
- Augmented Memorization Policy: Similar a la anterior, pero teniendo en cuenta un número arbitrario de pasos previos en la conversación.
- Form Policy: Es una extensión de Memorization Policy que se activa cuando se llega a una acción que implica el cumplimiento de un formulario.

Para cada paso de la conversación, se calculan las acciones predichas por cada política y el nivel de confianza en dichas predicciones y se elige realizar la acción para la que una política haya tenido mayor confianza. También se predefine un orden de preferencia de políticas para los casos en que dos acciones tengan la misma confianza. Por último, si ninguna predicción supera un cierto umbral predefinido, se puede utilizar la llamada *Fallback Policy*, que especifica una acción por defecto, normalmente indicando al usuario que no se ha entendido su intención, o la *Two-Stage Fallback Policy*, que pide al usuario que reformule su petición y en base a eso se vuelve a repetir el proceso de calcular la acción que se debe tomar.

Existen también algunos parámetros que especifican comportamientos del entrenamiento y la toma de decisiones como:

- Max History: Determina el número máximo de etapas del diálogo que el modelo tiene en cuenta para decidir.
- Data Augmentation: Con esta política se le indica al modelo en qué medida se puede aumentar el número de casos de entrenamiento mediante la combinación de diferentes historias.

Capítulo 4

Herramientas de desarrollo, tecnologías y metodología de trabajo

4.1. Herramientas de desarrollo empleadas

En esta sección se detallan las herramientas empleadas por el equipo de desarrollo durante este proyecto, tanto para la implementación del código, como aquellas herramientas que hayan servido también a la organización de las tareas, el mantenimiento del repositorio Git, y actividades de mejora del trabajo.

4.1.1. Herramientas para el trabajo online

- **Github**[1]: usado para subir el repositorio y mantener un historial con los cambios realizados en el proyecto por cada uno de los integrantes del equipo. Su sistema de *issues* ayuda a su vez a la comunicación con los directores del proyecto acerca de problemas y dudas.
- **Trello**[2]: hacemos uso de esta herramienta para mantener un tablero similar al usado en las metodologías ágiles, con las distintas tareas a realizar, para poder así asignarlas entre miembros del equipo y tener una progresión de las mismas de manera visual.
- **Overleaf**: herramienta empleada para la escritura de la memoria del proyecto. Funciona mediante LaTeX, y permite la edición del documento de la memoria almacenado en la nube, entre los distintos miembros del equipo.

4.1.2. Editores de Texto

- **Visual Studio Code**[3]: editor de texto empleado por varios miembros del equipo para el desarrollo del código del front end y el back end de la web de *Janet*. Posee una integración con Git, lo cuál facilita el manejo del repositorio, y de un sistema de extensiones que pueden ser utiles para múltiples lenguajes de programación.
- **Atom**: editor de texto que posee múltiples similitudes a visual Studio y que fue empleado a su vez en el desarrollo del apartado del Servidor y otros apartados del

[1] (https://github.com)
[2] (https://trello.com)
[3] (https://code.visualstudio.com)

proyecto. Incluye a su vez integraciones con Git y Github.

- **Sublime Text**[4]: editor de texto de tamaño ligero, que es de mucha utilidad en el caso de querer realizar pequeñas modificaciones a un archivo sin necesidad de abrir el proyecto completo.

4.1.3. Herramientas empleadas en los Clientes Móviles

Durante el desarrollo de la versión Web se realizó una serie de cambios tanto a la interfaz como la lógica de la aplicación. Estos cambios, añadidos a las nuevas funcionalidades de las que disponía la aplicación de *Janet*, hicieron necesaria la modificación de los clientes móviles, tanto *Android* como *iOS*.

- ***Android Studio***[5]: editor empleado para la modificación de la aplicación de Android. Está basado en *IntelliJ* y posee un emulador integrado que permite probar la App directamente mientras se van realizando los cambios.
- **Xcode**[6]: editor empleado para el desarrollo y la modificación de la versión de iOS en lenguaje *Swift*. Es necesario disponer de un dispositivo *Mac* para poder usarlo.

4.1.4. Otras herramientas empleadas

- **PuTTY**: este programa es empleado para acceder al contenedor virtual en el que se encuentra el servidor. Facilita su acceso a partir de dispositivos que emplean Windows como sistema operativo. Las modificaciones de los ficheros dentro del contenedor se realizaron principalmente con la herramienta *nano*.
- **VirtualBox**: ante la necesidad de tener que realizar múltiples pruebas de la instalación de *Janet*, se optó por emplear este programa, para así tener distintas instalaciones en sistemas virtualizados, sin tener que realizar formateos de discos reales.
- **WinSCP**: actúa de manera similar a PuTTY, ya que permite acceso al contenedor. Su principal diferencia radica en que WinSCP facilita mucho la subida de archivos.
- **Insomnia**: herramienta empleada por el equipo, principalmente durante la fase final del proyecto, para realizar *HTTP request*, simulando ser peticiones realizadas por cualquiera de los clientes. Se utiliza para comprobar el correcto funcionamiento en caso de situaciones donde faltara información o que pudieran dar lugar a error.
- **Narrador de Windows 10**: se trata de una herramienta que está incluida dentro del sistema operativo Windows 10. Se emplea para comprobar el correcto funcionamiento de la lectura de textos y etiquetas de HTML para personas con problemas visuales, con el propósito de que se cumpla el estándar de ARIA.

4.2. Tecnologías

En esta sección vamos a discutir las diversas tecnologías que se barajan para la consecución del desarrollo, junto a las razones por las que se han elegido las que utilizaremos. Muchas de las tecnologías que se emplean se heredan del proyecto anterior, por lo que para evitar rehacer código, las opciones quedan más limitadas.

[4](https://www.sublimetext.com)
[5](https://developer.android.com/studio)
[6](https://developer.apple.com/xcode/)

4.2.1. Desarrollo Web

Analizaremos las tecnologías que vamos a utilizar para hacer el cliente web de *Janet*. Este cliente se hace desde cero, por lo que las elecciones de tecnología se hacen de forma libre.

4.2.1.1. Back end

Para realizar el cliente, hemos decidido utilizar un framework web, ya que esto facilita mucho el desarrollo y nos permite conseguir código de mayor calidad con mayor facilidad. Algunas de las opciones que barajamos utilizar son:

- **Laravel:** Un framework muy completo de PHP, que otorga grandes facilidades a la hora de crear una página web, con un modelo vista-controlador bien definido. Uno de los miembros del equipo ya lo había utilizado para crear una API, pero nunca con una interfaz gráfica, por lo que esto podría suponer una mayor dificultad.

- **Bottle:** Se trata de un micro-framework para Python, diseñado para ser rápido, sencillo y ligero. Es una de las opciones ya que el framework se utiliza para el servidor de *Janet*, y aprender a utilizar este framework ayudaría a la comprensión del funcionamiento del servidor.

- **Flask:** Este micro-framework es uno de los más utilizados para Python, y está diseñado para poder empezar a utilizarlo fácil y rápidamente, manteniendo su habilidad para escalar sobre aplicaciones más complejas. Es similar a Bottle, y tiene la ventaja de tener más documentación, al ser el más usado.

Una vez discutimos qué framework íbamos a utilizar para hacer la web, elegimos **Flask** por los siguientes motivos:

- Como ya ha sido mencionado, es el más popular en la actualidad. Esto significa que es más fácil encontrar documentación, y que será más útil en el futuro profesional.

- Los directores del proyecto no recomiendan el uso de PHP. Entre otros motivos, PHP es más lento, y hay que escribir más líneas de código para hacer lo mismo que con Python.

- Es similar a Bottle, por lo que utilizarlo también ayudará a conocer este otro framework.

- Tenemos experiencia con él, y como consecuencia el desarrollo será más rápido.

4.2.1.2. Front end

Para desarrollar la interfaz web decidimos emplear un framework para poder modificar el estilo de la página con una mayor facilidad y obtener un acabado más profesional. Entre las diferentes opciones que barajamos para su empleo en nuestra web están:

- **Bootstrap:** se trata de uno de los framework web más utilizados del mundo, diseñado para facilitar el desarrollo *responsive* de la página web mediante un sistema de filas y columnas. Varios de los miembros del equipo ya disponían de experiencia previa en su uso.

- **React.js:** se trata de un framework web bastante completo para desarrollo de interfaces web, su funcionamiento se basa en la subdivisión de la página HTML en diferentes componentes. Dispone de una extensa documentación tanto en español, como en inglés.
- **Materialize:** es un framework web basado en el diseño de *Material Design* desarrollador por Google. Tiene la ventaja de disponer de una serie de estilos y animaciones predeterminados que facilitan que la app desarrollada disponga de un acabado profesional.

Tras una extensa evaluación de las diferentes posibilidades de framework de front end a utilizar, decidimos elegir **Bootstrap** por los siguientes motivos:

- Al tratarse de uno de los frameworks más empleados en la actualidad, se dispone de una extensa documentación que nos puede resultar muy útil, tanto en inglés como en español.
- Varios miembros del equipo ya habían hecho uso de Bootstrap con anterioridad, esto nos permitiría que avanzaramos de manera más rápida en el desarrollo, y que pudiéramos solventar las dudas con una mayor facilidad.
- Su sencillez y facilidad de implementación hará más rápido el desarrollo para una página no muy grande como es el cliente web de *Janet*.

Dentro de los elementos a usar para el desarrollo del front end, también disponemos de una librería de iconos open-source, FeatherIcons[7]. Se puede utilizar en diferentes botones de la interfaz para mejorar su aspecto visual.

Además de esto, se hará uso de la tecnología de Javascript *JQuery*[8], que permitirá modificar los elementos de la interfaz Web de manera dinámica, sin la necesidad de refrescar la página o de realizar peticiones POST o GET. Esta tecnología se empleará principalmente para la actalización de los mensajes mostrados por pantalla.

4.2.1.3. Bootstrap

Bootstrap (Ludo, 2019) es un framework CSS de diseño Web, originalmente desarrollado para la Red social Twitter, que facilita la estructuración de los elementos de la página web haciendo uso de un diseño basado en cuadrículas. Este diseño facilita la adaptación de diseño responsive (para distintos tamaños de pantalla) y para dispositivos móviles, dado que no existe la necesidad de especificar tamaños exactos para los elementos HTML. Es uno de los Frameworks CSS más usados del mundo, y actualmente se encuentra en su cuarta versión.

Existen dos métodos posibles para añadir Bootstrap a nuestro proyecto:

- **Descargar el archivo CSS de la página oficial y añadirlo a la carpeta de nuestro proyecto.** Este primer método posee la ventaja de que los estilos CSS van a cargar siempre que nuestra página sea accesible, pero presenta el inconveniente de que el tiempo de carga puede ser mayor. Esto es debido a que es necesario enviar el archivo completo al cliente, para poder cargar todos los estilos.

[7](https://github.com/feathericons/feather)
[8](https://jquery.com/)

- **Referenciar a la URL del archivo de la página de Bootstrap en la cabecera HTML de nuestra Web.** Este método es el más utilizado, dado que no es necesario que nuestro servidor envíe el CSS de la página. Posee el inconveniente de que si el servidor de la página de Bootstrap se cae, los estilos Bootstrap de nuestra página dejarían de funcionar.

Bootstrap implementa una gran cantidad de clases CSS que para ser referenciadas solo necesita añadirse en la etiqueta "class" de los elementos HTML a los que se les quiera aplicar. Las principales clases que implementan son las filas (row) y las columnas (col) que funcionan de tal manera que al declarar una clase "row" en su interior disponemos de un total de 12 columnas en las que podemos introducir nuestros elementos de manera horizontal. La principal ventaja de usar el sistema de columnas reside en que las columnas se adaptan al tamaño de la pantalla, lo cual hace más sencilla la visualización desde dispositivos móviles.

Además de eso se incluirá la librería de js de Bootstrap, junto con el plugin Popper.js[9], que emplean JQuery para funcionar. Estas librerías permitirán el uso de diversos plugins de Popper.js modificados por Bootstrap, que pueden resultar útiles durante el desarrollo del front end de la Web, y hacerla más atractiva visualmente.

4.2.1.4. Reconocimiento de voz

Para realizar el reconocimiento de voz web, tenemos que grabar la voz del cliente para poder procesarla y convertirla a texto. Esto supone varios problemas, entre ellos que no todas las personas disponen de un micrófono o que el navegador no sea capaz de reconocerlo. Para sortear esto, se ha decidido detectar desde la aplicación si no existe un micrófono, o si lo tiene pero la web no tiene permisos para acceder a él. De suceder esto, al pulsar el botón de grabar aparecería un mensaje en el navegador para informar del problema al usuario de que para emplear la funcionalidad, es necesario disponer del micrófono.

Con el fin de realizar la grabación de audio en el navegador, nos basamos en una librería de Javascript llamada WebAudioRecorder[10], la cual nos permite además de grabar audio, transformarlo en un archivo de sonido de con codificación WAV, OGG o MP3. Esto último es necesario para poder almacenarlo y después enviar este audio a una aplicación que nos permita extraer el texto detectado.

Para transformar la voz a texto tenemos dos posibilidades:

- Hacerlo desde el **front end**: Dejar que el código cliente sea el que traduzca a texto y se proceda a enviar este texto a *Janet*, como se haría habitualmente.
- Hacerlo desde el **back end**: Esta opción trataría de enviar el archivo con la grabación al servidor, y dejaría que fuera éste quien pasase la voz a texto, para acto seguido enviarle la respuesta de *Janet*.

Valoradas ambas opciones, optamos por dejar que el procesado lo haga el back end. Esto se debe a que hemos podido comprobar que las diferencias del navegador en el cliente generan incertidumbre sobre si pueden llegar a ejecutarse funcionalidades más avanzadas, como es

[9] (https://popper.js.org/)
[10] (https://github.com/higuma/web-audio-recorder-js)

la transcripción de audio. Sin embargo, del servidor se tiene total control, y se asegura que siempre se pueda realizar esta acción.

Para implementar la traducción en el servidor, se necesita una librería que nos permita hacerlo, ya que supondría un esfuerzo exhorbitado crear esta funcionalidad por nuestra cuenta. Para ello utilizamos la librería de Python SpeechRecognition[11], que permite obtener la transcripción de un audio. Esto ofrece compatibilidad con la librería utilizada en la web, WebAudioRecorder, y permite introducir el audio grabado por ella en en el servidor y transcribirlo. Además, esta librería puede utilizar, entre otras, la API de reconocimiento de voz de Google. Utilizar la API de Google garantiza la fiabilidad y calidad de las transcripciones.

4.3. Metodología de Trabajo

La metodología empleada a la hora de realizar el trabajo ha estado principalmente basada en los modelos de *Metodologías Ágiles* empleados en el desarrollo software. Esto se debe al hecho de ser un equipo pequeño de trabajo, y a la necesidad de modificación de los requisitos y tareas a medida que iba evolucionando el proyecto. La decisión de emplear una metodología ágil daba la posibilidad de que cada miembro del equipo pudiera adaptarse con mayor flexibilidad al ciclo de trabajo, y al empleo de herramientas propias de este tipo de metodologías, como el "tablero kanban".

Se emplea la herramienta *Trello* para realizar una organización y un seguimiento de las tareas a desarrollar en forma de tablero. Estas tareas tras ser definidas, son añadidas y categorizadas por tamaño. Dentro del tablero se dispone de una serie de columnas que indican el estado de las tareas que se encuentran en ellas. Las tareas comienzan en la columna de *Por hacer*, y van avanzando por las distintas columnas hasta que se completan. Una vez finalizadas, se pasa a las siguientes hasta completar el máximo número posible de tareas.

Se mantuvieron reuniones presenciales con los tutores cada dos semanas, en las cuáles se les cuestionaban acerca de las dudas y problemas que iban surgiendo durante el avance del proyecto. Estas reuniones se mantuvieron de forma telemática durante el periodo de aislamiento del estado de alarma del Covid-19.

El equipo de trabajo mantenía a su vez un canal de comunicación continuo a través de aplicaciones de mensajería instantánea, correos electrónicos y vídeo-llamadas, con el fin de evaluar los avances y las próximas tareas a realizar. Estas reuniones entre los miembros del equipo fueron haciéndose más frecuentes a medida que se acercaba la fecha de entrega.

[11] (https://pypi.org/project/SpeechRecognition/)

Capítulo 5

Descripción del Trabajo

5.1. Visión general

Como se ha mencionado en la sección 3.3.1, en el momento de comenzar a ampliar *Janet*, ésta disponía de dos clientes: una aplicación para iOS y otra para Android. Ambos tenían la capacidad de gestionar consultas relativas a disponibilidad de libros, horarios y ubicaciones de la biblioteca. Sin embargo, a simple vista, se ven ciertos puntos a mejorar:

- **Inconsistencia en el apartado visual:** pese a que las aplicaciones móviles para ambos sistemas operativos disponen prácticamente de las mismas funcionalidades, las interfaces gráficas son muy distintas la una de la otra, no pareciendo así que tengan relación entre si.
- **Necesidad de disponer sí o sí de la App:** es cierto que el uso de los Smartphones está ampliamente extendido. Sin embargo, la obligación de instalar un cliente móvil puede suponer una barrera de entrada para ciertas personas, ya sea por imposibilidad de descargarla por capacidad del teléfono, o por simple desidia.
- **Imprecisión en las respuestas:** en ciertas ocasiones *Janet* no entiende la intención relacionada a una pregunta, o, aún entendiendo correctamente lo que se pretende, puede devolver información incorrecta.
- **Evidencia de que se trata de un bot:** cualquiera que vaya a utilizar *Janet* sabe que no hablará como un ser humano. Sin embargo, esto se hace aún más obvio debido a la poca diversidad de las respuestas sobre un mismo tema y a las pocas posibilidades de conversación distendida, o "chit-chat", fuera de las funciones de la biblioteca.
- **Falta de contexto:** para usar las funcionalidades de *Janet* es necesario saber de antemano para qué puede ser usada, ya que no hay ninguna manera, salvo prueba y error, de descubrirlas.

Además, una vez comenzado el desarrollo, se van encontrando otros problemas que se añaden a los anteriores:

- Los instaladores no funcionan, siendo necesario realizar cambios en su configuración antes de ser empleados. Además, al usarlo más de una vez se producen errores en distintos puntos de la ejecución, debido a que se intentan realizar acciones que ya han sido ejecutadas.

- Ciertas librerías instaladas mediante *pip* pueden no existir dependiendo de la versión de *Python* asociada a éste en el sistema.

- El código es poco legible, habiendo pocos comentarios y siendo los nombres de las variables y métodos poco descriptivos. Esto obstaculiza su comprensión y dificulta hacer modificaciones o ampliaciones.

- Uso de una versión de Rasa aún en beta, la 0.13.0. Esto se traduce en una baja documentación asociada, gran cantidad de métodos obsoletos y bastantes bugs derivados del uso de otras librerías también desactualizadas.

5.2. Arquitectura

La arquitectura consta de tres partes diferenciadas que se comunican entre sí, utilizando servidores HTTP. En la actualidad, los 3 se hospedan en un mismo servidor, utilizando simplemente diferentes números de puerto para comunicarse.

5.2.1. Clientes

Nuestra aplicación consta de 3 clientes diferenciados, uno para web, otro para Android, y otro para iOS. Los clientes son las aplicaciones encargadas de interactuar con el usuario.

Estos clientes son capaces de obtener los mensajes que el usuario utilice para hablar con el bot, ya sea por voz o introduciendo el texto. Además de esto, es capaz de mostrar la respuesta al mensaje, ya sean imágenes, enlaces, texto, o incluso mapas. Los clientes son, básicamente, una aplicación de chat.

Sin embargo, el cliente no es el encargado de generar la respuesta a los mensajes. Este cliente envía el mensaje del usuario, transformado a texto si es que ha sido introducido por voz, a otra parte de la aplicación. Esta parte es la que se conoce como el servidor de *Janet*.

5.2.2. Servidor de Janet

Este servidor es la parte central de la aplicación. Es el que recibe los datos de entrada, se los envía directamente a Rasa, y elabora la respuesta correcta.

Cuando este servidor analiza la respuesta que ha obtenido de Rasa, puede saber cuál es la intención del usuario. Según sea esta intención, si se desea un libro, tiene que buscarlo en la mediante las APIs de WorldCat (vistas en 3.1.1). Tan solo necesita saber qué es lo que busca el usuario, y eso será extraído por Rasa. A continuación, se puede obtener los datos del libro para poder otorgarle la información al cliente.

La conexión con Rasa se hace desde Python, tratándolo como cualquier otra librería. Tan solo se necesita el modelo del lenguaje guardado, que cargará la respuesta que debe tomar ante cada entrada de usuario. Para ejecutar algunas acciones ante ciertas entradas en Rasa, es necesario acceder a otro servidor, el servidor de acciones de Jarvis.

5.2.3. Acciones de Jarvis

El llamado "Acciones de Jarvis", no es más que un servidor de acciones de Rasa. Es el encargado hacer algunas acciones más complejas que una simple respuesta de texto al recibir ciertas entradas. Es necesario definir dichas acciones en un archivo y ejecutarlo en un servidor distinto.

Pongamos un ejemplo: la acción de saludar. Aunque en un principio pueda parecer que cuando el usuario salude el servidor debe responder devolviéndole el saludo, es algo más complejo. Se ha hecho que se guarde el nombre del usuario, y que el bot lo mencione en el saludo si es que lo conoce. La acción especial consiste en ver si se conoce el nombre del usuario: Si se conoce, se podrá crear una respuesta que lo mencione, pero si no, se tendrá que enviar una respuesta más genérica.

5.3. Bot

Janet es un bot cuyo objetivo es facilitar los accesos y utilidades de la bibliblioteca de la UCM a cualquier tipo de público. Para implementar este objetivo, se utiliza como base el software de código abierto Rasa, el cual permite crear un chatbot de una forma simplificada, y pudiendo lograr resultados óptimos.

Cuando este proyecto fue implementado originalmente, Rasa se encontraba en un estado de reciente creación. Es por ello por lo que la forma en la que se tenía que interactuar con la librería era compleja y poco intuitiva. Uno de los objetivos que se ha planteado es actualizar Rasa a una versión más moderna, ya que durante este último año se ha mejorado y simplificado mucho la implementación y el uso de esta herramienta.

Para actualizar la librería escogemos utilizar la versión 1.9.3, un gran salto desde la versión 0.13 que se utilizaba anteriormente. Es por esto por lo que se suceden muchos cambios y muchos de los aspectos que se utilizaban anteriormente quedan obsoletos. Entre los cambios más grandes que ha sufrido la librería encontramos:

- Rasa core y Rasa NLU eran dos proyectos distintos que se instalaban por separado. En la actualidad, han sido agrupados en uno, lo cual facilita tanto la instalación como su uso. Esto implica que la forma en la que se importa desde el código de Python también tiene que cambiar.
- La "Pipeline" que sigue Rasa ha cambiado. Esto es el conjunto de operaciones, transformaciones y políticas que se aplican a los datos de entrenamiento para obtener el modelo final entrenado. Esto quiere decir que habrá que rediseñar el "Pipeline" completamente para que pueda seguir funcionando de forma adecuada.

Al actualizar la versión de Rasa, podemos observar que el servicio conocido como "Jarvis" ha dejado de tener utilidad. Este servicio era el encargado de mediar entre el servidor que recibe las peticiones del chat con el usuario y Rasa. Se encargaba de recibir la petición del usuario, hacer varias transformaciones, y generar tanto una entrada como una salida que fuese útil para ser devuelta al resto del aplicativo.

Para hacer la función de Jarvis, el cual podemos eliminar, se utiliza directamente la aplicación de Rasa. Esto se ha hecho de dos formas diferentes durante el proyecto, y hemos

valorado las ventajas de utilizar cada una:

- Crear un servicio con el webhook que provee Rasa. Esta API proporciona toda la funcionalidad de Rasa mediante llamadas HTTP. Sin embargo, es necesario ejecutar un comando de Rasa y tener un servidor corriendo, el cual debe abrirse cada vez que se quiera ejecutar el resto de la aplicación.
- Utilizar la API para Python que proporciona Rasa. Esta API, aunque poco documentada, proporciona también la funcionalidad completa de Rasa desde código para Python. Esto permite eliminar el intermediario del servidor HTTP, ya que desde el propio código se puede cargar el modelo y mantener conversaciones con el bot.

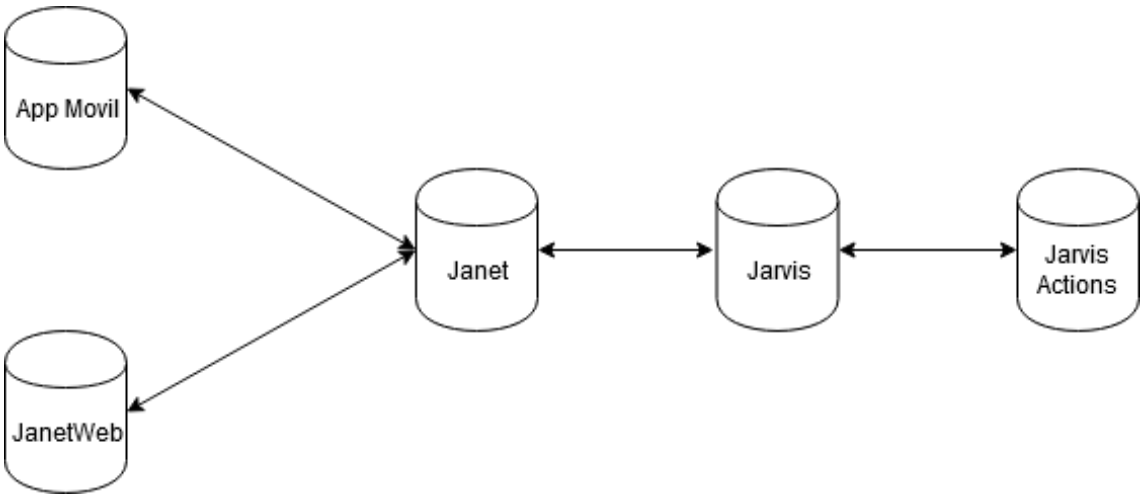

Figura 5.1: Esquema de las comunicaciones utilizando la API HTTP de Rasa

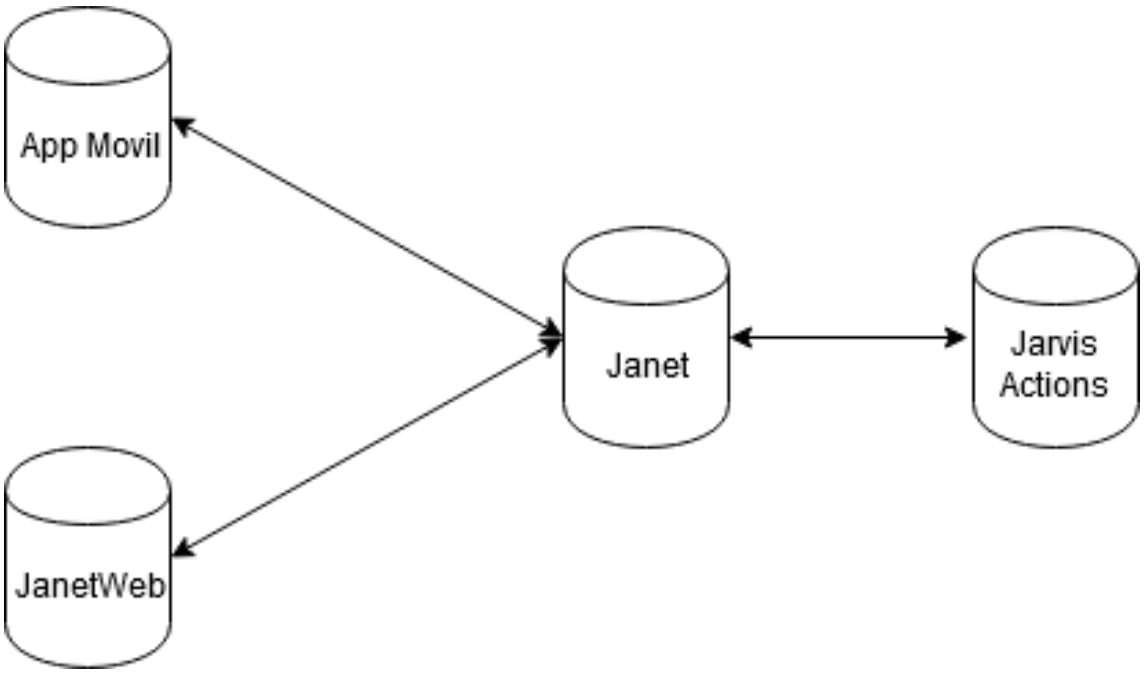

Figura 5.2: Esquema de las comunicaciones utilizando la API de Python de Rasa

Tras probar la opción de la API HTTP, escogemos utilizar la API de Python ya que se integra mejor con el resto de código, y nos permite tener un servicio menos, como se muestra en las figuras. Podemos observar que es necesario seguir manteniendo el servidor de JarvisActions. Este es una parte de Rasa, encargada de ejecutar las acciones personalizadas tras recibir un mensaje.

Toda la ejecución de Rasa se delega a *Janet*, la aplicación principal. Esta recibe mensajes del usuario, y ejecuta acciones de Rasa para predecir la intención del usuario y ejecutar las acciones adecuadas. Si la acción requiere de algo más que una respuesta, como podría ser el caso de la búsqueda de un libro, se envía a JarvisActions para que sea tratada.

5.3.1. Mejoras en el bot

Uno de nuestros propósitos era conseguir un bot mejor, que pudiese mantener una conversación más coherente con el usuario, y que pueda ayudar a los usuarios con sus necesidades en la biblioteca. Para lograr esto, se mejoran diversas partes del bot, las cuales se describen a continuación.

Para empezar, **se limpian las bases de datos**. Muchas de las bases de datos que utiliza el bot contienen entradas repetidas. Por ejemplo, en la lista de autores, aparecía varias veces el nombre de un mismo autor con diferente capitalización. Aunque esto no afecta de forma especial al bot, si que puede hacer que algunas búsquedas tarden algo más si el número de datos repetidos es excesivo. Aunque el impacto no sea excesivo, se borran igualmente, ya que no supone un esfuerzo grande.

Otro cambio importante ha sido incluir una **gran cantidad nueva de ejemplos de entrada para cada intent**. Dando más formas en las que el usuario puede expresar una misma idea al bot le ayudará a entender y reconocer las intenciones del usuario. Vamos a poner un ejemplo:

Durante el desarrollo, se dió la versión inicial de la aplicación a varios usuarios y se les pidió que realizasen una búsqueda de un libro por título y autor. Lo que se observó fue que la mayoría de usuarios introducían la información de la forma "Busca *(Título del libro)* de *(Nombre del autor)*". Sin embargo la mayoría de ejemplos de entradas de usuario en la aplicación era de la forma "Busca el libro titulado *(Título del libro)* del autor *(Nombre del autor)*". La segunda forma es mucho menos natural, y no es como un usuario suele comunicarse. Es así por lo que se introducen muchos más ejemplos que se asemejen a la primera forma, centrándonos en cómo se comunicaría realmente un usuario con un bot.

Cuando se han hecho pruebas con amigos y familiares, se ha observado que lo primero que suelen hacer son preguntas no relacionadas con el dominio del bot. Si un usuario pregunta algo y recibe una pregunta errónea o por defecto, puede dudar de la calidad del chatbot. Para evitar esta situación, **hay que hacer que responda coherentemente a los temas que un usuario suela preguntar típicamente y estén fuera del dominio**.

El bot existente ya respondía a varios temas distintos a los libros como saludos, dar las gracias, preguntar por el nombre. Aun así este debe responder a muchas más preguntas que los usuarios hace de manera bastante común (Kolodkin, 2017), y estas interacciones son las que se han desarrollado. Entre otras, ahora *Janet* sabe responder cuando le pre-

guntan cómo está, si le dicen que la quieren, a preguntas sobre su edad, contar chistes... Estas nuevas acciones no son vitales, pero logran dar una personalidad a *Janet*, y hacer que parezca menos un bot con salidas predeterminadas.

Las historias de usuario son flujos del diálogo que el bot puede seguir. De esta manera, sabrá cómo responder cuando el usuario le pregunta algo determinado, incluyendo el contexto actual de la pregunta. Estas historias en la versión original fueron hechas con la versión de rasa interactiva, en la que mientras se habla con el bot se le dice qué debe hacer ante cada entrada. Esta forma de introducir las historias tiene múltiples inconvenientes:

- El título de las historias creadas así es generado automáticamente. Este título no es indicativo de qué indica la historia, por lo que es costoso reconocer la intención de las historias y escalar el archivo.
- Se introducen slots innecesarios o con datos mal escritos. Esto quiere decir que, cuando se busca por ejemplo un libro por solo su título, a veces se introducen de forma errónea otros slots diferentes al título como el autor. Esto puede causar confusión en las historias y no deberían aparecer.
- Las historias creadas son demasiado cortas; contienen solo una intención y la respuesta del bot a dicha intención. Esto hace que el bot no pueda dar respuestas más complejas a una conversación, ya que solo tendrá en cuenta lo último que ha dicho el usuario.

Para arreglar estos inconvenientes, se reescribe el fichero de forma manual. De esta manera se puede dar títulos a las historias e eliminar los slots innecesarios de las historias existentes, como se puede observar en la figura 5.3. En esta figura observamos como se ha modificado la historia para que tenga título, y se elimina el slot duplicado para la persona.

Figura 5.3: Comparación de una historia generada frente a otra escrita manualmente

También son agregadas **nuevas historias** para las nuevas intenciones, y se crean algunos **flujos de diálogos más complejos**. Por ejemplo, un problema notable se daba en el siguiente flujo de diálogo: el usuario pedía un libro sin dar ningún título, y el bot le preguntaba cuál es el libro que desea. Entonces el usuario respondía con solo el título del libro, pero como el bot no seguía el flujo de más de un mensaje, no sabe qué pretende decir. Para arreglar esto se introduce una historia en la que si el usuario envía un título después de pedir un libro sin sus datos, el bot pasa a buscar el libro con ese título. La diferencia en el resultado se puede contemplar en la Figura 5.4.

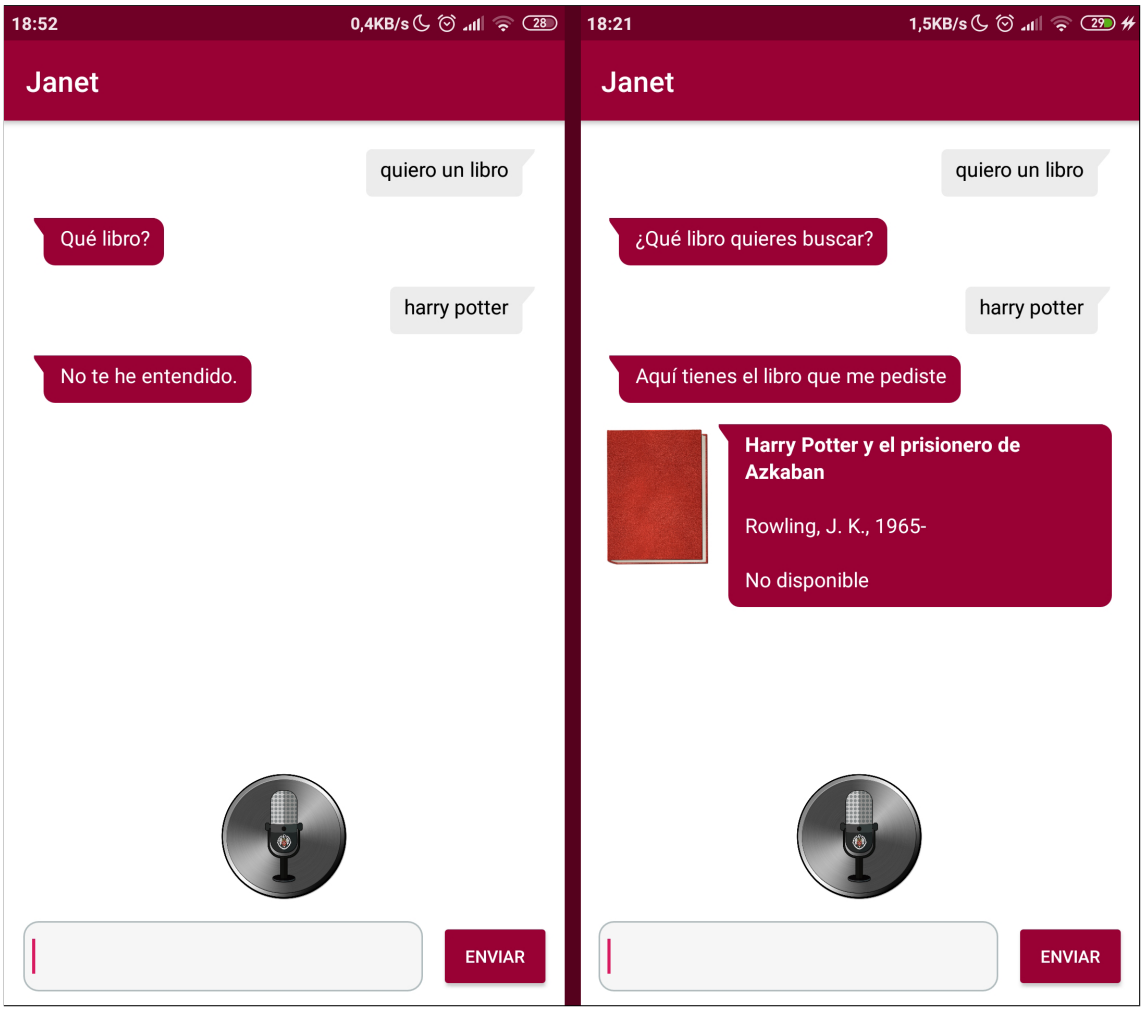

Figura 5.4: Comparación del flujo de historia antiguo, frente al flujo con historias complejas

Algo que no se podía hacer anteriormente con *Janet* es **obtener los correos electrónicos de las bibliotecas**. Sin embargo, dicha información ya se encuentra en la base de datos que se maneja, por lo que es sencilla de obtener. Para tratarla agregamos nuevos intents, junto con varias formas de preguntar por el correo, historias, respuestas... Lo más importante sobre esta aplicación es la accesibilidad. Es por ello por lo que se quiere que se pueda mandar un correo de forma instantánea una vez obtenida la información. Para eso se requiere que al pulsar sobre el correo en la aplicación cliente se abra un nuevo correo en la aplicación pertinente de cada sistema. Esto hace que se tenga que modificar el código de los clientes móviles.

Cuando el cliente móvil reciba una respuesta de la categoría "correo", al ser pulsada la dirección se abrirá la aplicación de correos que se tenga instalada. Se hace de igual manera en la web.

Finalmente, para mejorar el bot, también hay que fijarse en la **usabilidad**. Como un usuario no familiarizado con la aplicación no sabe cómo se interactúa con *Janet*, se necesita alguna forma de contarle a este qué puede hacer. Además de darle solución a este problema en los clientes como se contará a partir de las siguientes secciones, podemos hacer varias

cosas desde el funcionamiento del bot:

- Se ha agregado una intención de usuario para la pregunta "¿Qué sabes hacer?", y reformulaciones similares de la misma. De esta forma, si el usuario pregunta directamente esto, *Janet* podrá responderle con una lista de funcionalidades que soporta.
- Cuando el usuario pedía algo que el bot no estaba preparado para responder, este respondía con un simple "Lo siento, no te he entendido". En vez de esto, es conveniente que además de decir esto, el bot de información sobre qué sabe hacer para que el usuario no cometa este error de nuevo. De esta manera, se puede reconducir al usuario a un flujo de conversación correcto.

5.3.2. Legibilidad del código y de las consultas

Uno de los problemas encontrados para comenzar a trabajar fue la dificultad para entender el código, producida por la confluencia de la deficiente documentación de versiones antiguas de Rasa, ya mencionado anteriormente, y el uso de nombres de variables poco descriptivos junto a fragmentos de código sin comentar en los archivos correspondientes al servidor de *Janet*. Estos problemas se han ido paliando según se iban rehaciendo las partes de la funcionalidad mencionadas en el apartado anterior con la intención de que, en caso de que el proyecto sea continuado en años posteriores, su funcionamiento interno sea lo más sencillo posible de comprender.

Otro elemento que nos frenaba a la hora de añadir funcionalidades o solucionar fallos eran los logs. En un inicio, los logs se escribían en archivos, daban poca información del procesamiento interno y nula sobre los errores que pudieran suceder, y se sobreescribían con cada actualización de la herramienta, con lo que se perdían las interacciones que pudieran haber hecho previamente los usuarios. Para solucionar esto, se decide portar esta funcionalidad para ser gestionada mediante *journalctl*, que guarda un registro de todos los mensajes generados por un kernel. Con este cambio, la información de la que se quiere mantener un registro pasa a imprimirse y cualquier error producido en el servicio también queda registrado. Cabe destacar que fue necesario especificar la codificación de los mensajes a utf-8 por problemas de compatibilidad en Debian.

5.3.3. Comprensión y capacidad de responder en inglés

Otra mejora importante realizada es dotar a *Janet* con la capacidad de entender y procesar mensajes en inglés y hacerla capaz de también responderlos en dicho idioma. Esta cualidad se hace incluso necesaria en una comunidad internacional como puede ser la universidad complutense, donde más del 10 % de los alumnos son extranjeros[1] y para los cuales la exclusividad en español de una herramienta puede ser una barrera de entrada.

La intención inicial para la implementación era la disposición de botones en la interfaz web desde los cuales definir el idioma con el que se fueran a realizar las consultas y en el que se fuese a leer la respuesta. Éste es un enfoque bastante estándar en el que los botones de cada idioma suelen estar representados con banderas de los países representativos de dichas lenguas. Sin embargo, se presentan ciertos inconvenientes con la selección del idioma desde el front end:

[1] Informe del rector al Claustro UCM del 28 de febrero de 2018: `https://www.ucm.es/informe-rector-claustro-ucm-28feb18`

- El cambio requiere ser replicado en todos los clientes, ya sea cualquiera de las aplicaciones móviles o la página web.
- En caso de añadir en un futuro funcionalidad en más idiomas, puede llegar a ser poco escalable para pantallas pequeñas y hacer falta un rediseño de la forma de elección.
- Se corre el riesgo de dejar obsoletas versiones antiguas de las aplicaciones en las que no se proporciona el idioma.
- La cantidad y el formato de la información transferida entre clientes, front end y back end se complica.

Es por esto que se descarta este enfoque en pro de uno que no requiere de la activación expresa del usuario y el cual se puede llevar a cabo a nivel de servidor. Finalmente se acaba utilizando la librería *langdetect*[2], un port de Java a Python de la librería language-detection de Google. Con esta librería, justo antes de insertar el mensaje en el agente de Rasa, se analiza el mensaje del usuario y se estima el lenguaje en el que está. Si se detecta español o inglés, se pasa a los respectivos agentes para dichos idiomas. Sin embargo, puede darse el caso de que un mensaje sea muy corto o ambiguo y no se reconozca con total seguridad el idioma, en cuyo caso, será procesado de manera predeterminada por el agente en español.

Como se acaba de mencionar, cada idioma tiene un agente diferente asignado. Sin embargo, ambos comparten las mismas intenciones que pueden predecir, conversaciones sobre las que se han entrenado y tipos de acciones que se pueden realizar y respuestas que se pueden dar. Para la implementación del agente en inglés, se traducen las frases de entrenamiento asociadas a cada intención así como las respuestas correspondientes.

Un elemento que comparten es el servidor en que se ejecutan las acciones personalizadas. Con respecto a este componente, se encontró un problema a la hora de gestionar situaciones inesperadas: ciertos mensajes de error en casos como ubicaciones no encontradas o fallos a la hora de buscar libros, estaban gestionados por este módulo de acciones, por lo que se respondía siempre en español independientemente del idioma de la petición. La primera solución que se sopesa es crear otro servidor de acciones en el que dichos mensajes estuvieran en inglés, pero esta solución complicaba la arquitectura de la aplicación, además que habría mucha funcionalidad duplicada. Finalmente, la solución que se acaba llevando a cabo es hacer un pequeño rediseño del funcionamiento de estas acciones, migrando funcionalidad de la gestión de errores hacia el conocimiento del dominio de Rasa, y siendo así estos mensajes independientes de la funcionalidad de la acción y gestionados por cada agente.

5.4. Diseño Web: Back end

Con la creación de una página web, se pretende servir la misma funcionalidad que los clientes móviles de forma más accesible. Para servir la web, se utiliza el microframework de Flask . Con él, se crearan las diversas rutas que necesitaremos. Como es una aplicación sencilla, solo necesitaremos tres páginas:

- La página principal, desde la cual se podrá chatear con *Janet* tanto por texto como por voz.

[2] (`https://pypi.org/project/langdetect/`)

- La página que nos permite leer y aceptar la política de privacidad.
- La página de información, que dispone de datos adicionales sobre *Janet*, así como enlaces a las Apps móviles y ejemplos de uso.

Con cada usuario creamos un identificador único, con el que se le va a poder identificar mientras acceda mediante el mismo navegador. Se ha escogido hacerlo así ya que sería necesario un registro de usuario para que se mantenga siempre el historial, pero al ser una aplicación sencilla muchos usuarios la dejarían de utilizar al enfrentarse a una pantalla que les haga introducir datos. Este identificador único se hace con una cadena de 130 caracteres alfanuméricos generada de forma aleatoria. Con esto, se asegura que la posibilidad de que dos usuarios distintos identificadores iguales sea despreciable.

5.4.1. Privacidad

La aplicación de *Janet* recoge todas las conversaciones que tiene con el usuario, junto a cómo ha respondido. Esto permite analizar los datos de conversaciones para averiguar cómo ha fallado el software, y poder proponer soluciones.

Al no existir ninguna forma de registro, la forma en la que se identifica a cada usuario es media un identificador pseudo-aleatorio. En ningún momento se obtiene directamente información que nos permita identificar físicamente a la persona tras un mensaje. Además, tampoco se registran las direcciones IP recibidas.

Esto no significa que el usuario no pueda incluir información personal en los mensajes que envíe en la aplicación. Entre otras cosas, *Janet* recoge el nombre del usuario si este lo dice para poder saludarle. Estos mensajes pueden hacer identificable a la persona, y según el **reglamento general de protección de datos**[3], se debe informar al usuario sobre el tratamiento que se harán con sus datos y pedirle el consentimiento expreso.

Con esta finalidad, la primera vez que un usuario entre a la página será dirigido hacia la política de privacidad. Si este no la acepta, no será posible proceder a comunicarse con el bot, ya que necesitamos consentimiento explícito. Este consentimiento se mantiene mientras no borre el historial, y se aprovechará la ocasión para asignarle el identificador único de dicho usuario.

5.4.2. Gestión del reconocimiento de voz

La grabación de la voz se hace desde Javascript. El usuario debe dar permisos a la página para permitir la grabación de su micrófono. Una vez el usuario presiona el botón de grabación, se obtiene el audio. Una vez el usuario vuelve a presionar el botón para detener la grabación, el audio, codificado en WAV, se hace una petición POST al servidor con el archivo generado.

En el servidor, utilizando la librería *SpeechRecognition* para enviar este audio al reconocedor de voz de Google. Utilizamos este porque nos ofrece una muy buena fiabilidad y unos resultados muy acertados. Una vez tenemos el texto transcrito del usuario, la web lo recibe y otra vez y lo escribe en pantalla. Acto seguido, podemos enviarlo igual que hacemos con

[3]Más información en https://rgpd.es

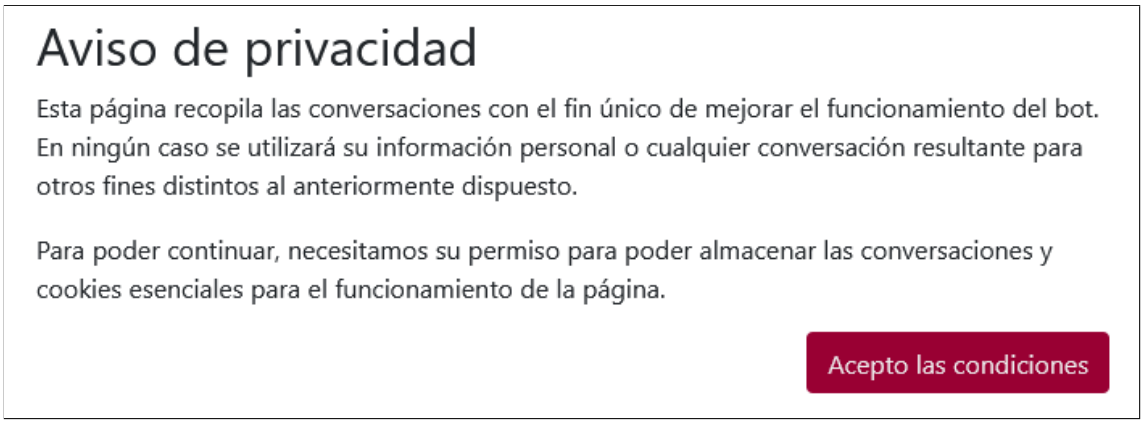

Figura 5.5: Aviso de privacidad en la web de *Janet*

el texto escrito en la página para que sea procesado y se obtenga una respuesta. Se vuelve a pasar por el cliente con la finalidad de que aparezca el texto en cuanto haya sido transcrito en lugar de tener que esperar a que también se haya obtenido la respuesta de *Janet* a su vez.

Cabe destacar que el reconocimiento de voz en el navegador solo ha sido comprobado para Firefox y Google Chrome. Se debe principalmente a dos razones:

- Chrome es el navegador más ampliamente utilizados, y Firefox también cuenta con una amplia base de usuarios. Con estos dos se cubren la amplia mayoría de usuarios web.

- Ambos navegadores ofrecen muchas facilidades en sus implementación para acceder a la grabación de voz de forma sencilla.

5.4.3. Gestión del error 404

Actualmente la web consta de de un total de 3 ventanas accesibles mediante peticiones GET: la ventana de las condiciones de privacidad, la ventana principal, y la ventana de información. Es por ello que al intentar acceder a cualquier otra dirección que no sea "/", "/info" o "/privacy" se produce el lanzamiento del error 404 "Not Found" mostrando por defecto una ventana provista por el sistema que aloja la Web. Para evitar que esto se produzca existen principalmente dos opciones:

- **Se crea una vista personalizada para el error 404** con un mensaje descriptivo sobre lo que ha sucedido. Este método es el más utilizado en páginas web de mediana o gran envergadura.

- **Se realiza un redireccionamiento a la ventana principal**. Este método es más común en Webs de pequeño tamaño, que no disponen de gran cantidad de direcciones bajo su dominio.

Finalmente se decidió aplicar este segundo método, debido al bajo número de subdirecciones que posee la web de *Janet*. Para ello se implementó en el archivo *views.py*, que contiene las rutas de la web, un "errorhandler" que capturaría los 404, redirigiendo al usuario a la dirección principal de la web en caso de acceder a una dirección no existente.

5.5. Diseño Web: Front end

El diseño del front end tiene una importancia especial dentro del proyecto, ya que se trata de la parte de la web con la que los usuarios van a interactuar directamente. Es por ello, que se debe diseñar una interfaz que no resulte compleja, y que sea intuitiva para poder ser utilizada por cualquier persona, sin requerir de ningún tipo de manual.

5.5.1. Evolución del diseño

El diseño del front end de nuestra página Web pasó por diferentes etapas hasta llegar a la versión final.

Primera versión de la Interfaz Web

Cuando comenzamos con el diseño de la interfaz Web se tenía claro que lo primordial era lograr el correcto funcionamiento del back end, y que las respuestas de *Janet* se envíen correctamente (como ya sucedía en las versiones para Android y iOS). Es por ello que la primera versión de la aplicación web tenía un diseño muy simplista, con la intención de asegurarse de que el sistema de mensajes funcionaba correctamente, antes de implementar un diseño definitivo. En esta versión, la página hacía un uso muy básico de bootstrap, y gran parte de los estilos y el posicionamiento de los elementos estaba realizado directamente con CSS.

Sin embargo, la estructura HTML desarrollada en esta versión se seguiría empleando como base para las versiones posteriores. La página consta de una cabecera con el nombre de *Janet*, una zona donde se muestran los mensajes enviados y recibidos, y un formulario de entrada de texto donde escribir los mensajes que se quieren enviar. una vez pulsado el botón de enviar se ejecutaba la función de envío de datos a *Janet* en JQuery, que se encargaba de enviar la información al servidor de *Janet*, y de insertar los mensajes recibidos de forma dinámica.

La página también disponía de un menú de opciones, para implementar posibles funciones y que nunca llegó a ser implementado. Estos defectos serían subsanados en las siguientes versiones de la aplicación, basándose para ello en criterios de diseño más específicos.

Segunda versión de la Interfaz Web

Una vez asegurado que el envío de mensajes, y que *Janet* mostraba las respuestas adecuadas, nos centramos en trabajar más a fondo en la mejora del diseño Web. Para ello hicimos un uso más completo de Bootstrap.

Se debía lograr que la interfaz estuviera correctamente dividida en la diferente proporción de filas y columnas. Asegurándose a su vez de que el diseño que se estaba realizando fuese "responsive", es decir, que se adaptara a diferentes tamaños de pantalla sin deformarse.

Para ello aplicamos los *Principios de usabilidad de Jakob Nielsen* (Allas Miguelsanzr, 2017). De esta manera, la información innecesaria es eliminada de la pagina y la estructura se limita a la cabecera, la zona del chat, y un recuadro de texto para escribir los mensajes. También se modificaron los estilos CSS para que se siguieran ciertos convenios estilísticos

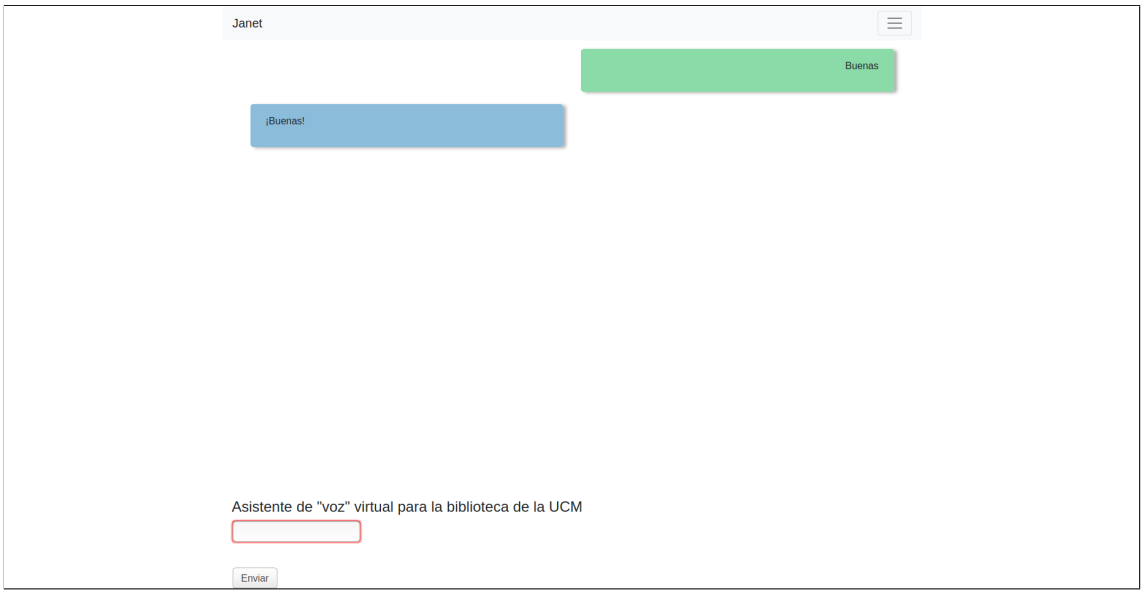

Figura 5.6: Primera Versión de la Página Web

(botón de envío de color verde, banner que ocupe el ancho completo de la pantalla).

Aún con los nuevos cambios visuales, la interfaz seguía presentando una serie de fallos. La adición de la funcionalidad de detección de voz, supuso la introducción de un botón nuevo que no se encontraba integrado con el resto del diseño de la aplicación, seguía existiendo el botón de menú de opciones, y muchos de los colores de la página no seguían los principios de minimalismo por los que aboga Nielsen.

Figura 5.7: Segunda Versión de la Página Web

Tercera versión de la Interfaz Web

Con esta última versión de la aplicación, se trató de pulir el modelo seguido con la segunda versión, extendiendo el uso del minimalismo, y el uso de iconos en vez de texto para los distintos elementos de la interfaz, tal y como viene descrito dentro de los principios de *Nielsen*.

Gran parte de las mejoras de usabilidad fueron realizadas durante el desarrollo de esta última versión, arreglándose a su vez diversos errores que se habían cometido durante las versiones anteriores. Durante esta fase, el equipo aprendió formas más eficientes de emplear algunas de las clases de Bootstrap, lo que hizo necesaria la reestructuración de varios elementos de la web, que no se encontraban declarados correctamente.

Tras el desarrollo de esta última versión se dio por acabada la interfaz web de la aplicación, habiendo implementado nuevas funcionalidades y consiguiendo un aspecto visual similar al de otras aplicaciones de la UCM. Estas mejoras plantearon la posibilidad de modificar los clientes móviles para que las versiones dispusieran del mismo aspecto y funcionalidades, cambios que finalmente se acabaron realizando.

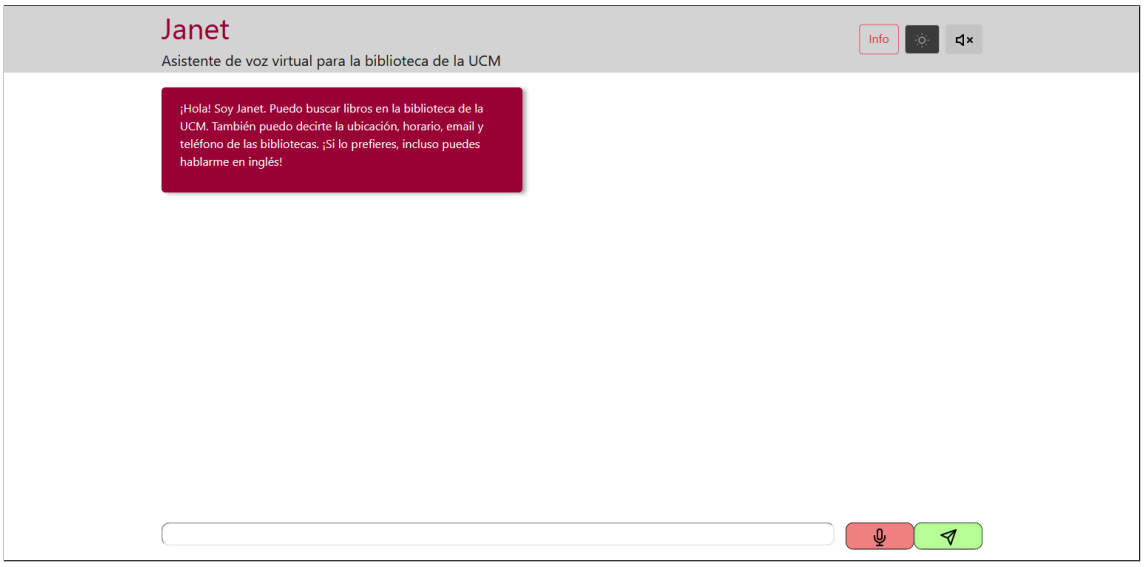

Figura 5.8: Tercera Versión de la Página Web

5.5.2. Mejoras de usabilidad y estilos

Durante el desarrollo de la última versión Web se recibió "feedback" y consejos acerca de posibles mejores funcionales y visuales. Se sugirieron y se estudiaron deferentes ideas, que acabaron materializándose en una serie de cambios en la interfaz:

- El botón del menú de opciones fue eliminado, ya que no disponía de una funcionalidad concreta.
- Se añadió un sonido de notificación, que se reproduce cada vez que el usuario recibe un mensaje de *Janet*. Este sonido puede resultar especialmente útil para personas con problemas de visión.

- El botón del grabación se movió a la zona inferior junto con el de envío del mensaje, ambos botones cambiaron los estilos y el texto que poseían por un icono representativo de su función para ajustarse a los criterios de *Nielsen*. A el botón de grabación se le dispuso de una animación CSS para indicar cuando está grabando, y de un *Popover* para indicar que se debe volver a pulsar el botón para finalizar la grabación.

- En las primeras versiones anteriores de la aplicación si el micrófono no se encontraba disponible el botón quedaba inutilizado al pulsarlo, esto podía resultar confuso a los usuarios que intentaran enviar un mensaje con él. Por ello se modificó el código JQuery para que si se pulsa el botón sin tener acceso al micrófono se genera una *alerta* informando al usuario de ello.

- Al emplear la grabación por voz, si esta no detectaba palabras en el mensaje no daba ningún indicativo sobre ello, lo que podría confundir al usuario. Por ello, se añadió la condición de que si esto sucede, la aplicación genera un mensaje en pantalla informando sobre ello que se va desvaneciendo poco a poco.

- Se añadió un botón que implementaba la función para que *Janet* leyera los mensajes mediante un sistema *TextToSpeech*. Dicho sistema permitiría más tarde, tras su implementación en el servidor, la lectura de mensajes recibidos en inglés.

- Durante el desarrollo de la última versión, se modificaron los colores de la Web con el fin de que tuviera un diseño más minimalista, en concordancia con otras aplicaciones de la UCM. Para ello se aplicó de manera libre el código de colores empleado por la Universidad Complutense[4], adaptando los colores de los rótulos, y mensajes a un formato similar al descrito en dicha página.

- Se añadió un botón para poder cambiar el estilo de la página al modo de alto contraste, similar al de la versión de los clientes móviles.

- Las respuestas de libros de *Janet* aparecían ahora con la portada del libro en cuestión, empleando para ello el mismo método usado para los clientes móviles.

- Se creó una ventana de información a la que se podía acceder a través de un enlace en la página principal. Dicha página contiene una pequeña descripción de *Janet* con ejemplos de uso, así como enlaces al repositorio en *Github* y los sitios de descarga para la Apps móviles.

- Se creó un archivo *manifest.json* permitía, a través de la implementación de un acceso directo en las versiones móviles, que la web se viera de forma *standalone*. Esto implica que se vea sin la barra de navegación del navegador, similar a como sería una aplicación nativa para el móvil, como se puede apreciar en la Figura 5.9. Este sistema hace que durante la apertura en *standalone* se disponga de una ventana de carga, junto con el logo de la aplicación.

5.5.3. Accesibilidad dentro de la Versión Web

Es importante facilitar el uso de la Web para el mayor número de personas posible, es por ello que durante el desarrollo de la web se realizaron diversos cambios, para facilitar la accesibilidad para las personas con dificultades visuales.

[4]https://ssii.ucm.es/colores-y-tipografia

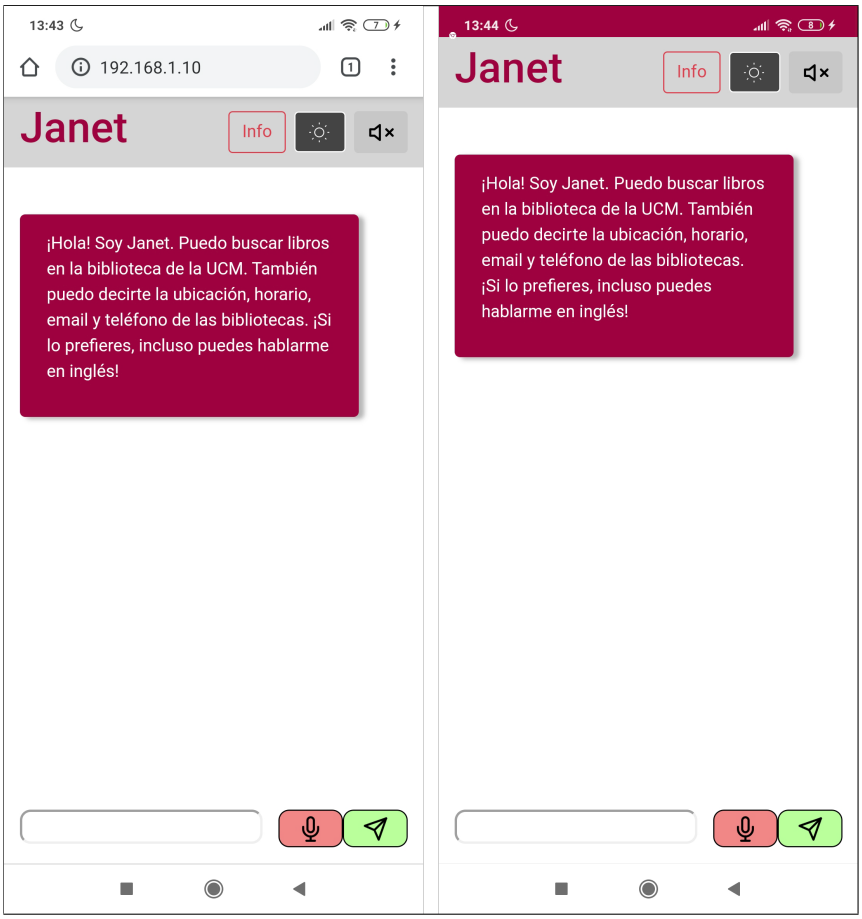

Figura 5.9: Comparación entre *Janet Web* en el navegador y *Janet Web* en modo *Standalone*

Modo de alto contraste

Como se ha mencionado previamente en la lista de mejoras, uno de los cambios fue la adición de un modo de alto contraste. Para ello, se emplea de base el modelo implementado en las versiones móviles, de tal manera que al pulsar el botón, se ejecuta una acción en JQuery para añadirle la clase CSS de alto contraste a los elementos que hiciera falta. El estilo del modo de alto contraste para la mayoría de estos elementos se limita principalmente a cambiar los fondos a negro, mientras que los textos y otros elementos cambiaban a color blanco, con alguna excepción puntual.

Implementación de atributos ARIA

El otro cambio importante que se realizó para mejorar la accesibilidad en la Web, fue el empleo de atributos **ARIA** (*Accessible Rich Internet Applications*) (MDN, 2020). Las personas con discapacidad visual suelen hacer uso de herramientas tales como los narradores de voz para poder describir la información y funcionalidad de los elementos en pantalla. Estos narradores además de leer el texto que se muestra visualmente en la página, leen las etiquetas y atributos de elementos HTML, tales como los títulos, botones y enlaces, permitiendo dar información adicional acerca del mismo.

Para comprobar como leen estos sistemas de narración nuestra web, se emplea el narrador de Windows 10, implementado dentro del mismo sistema operativo. Usando las flechas

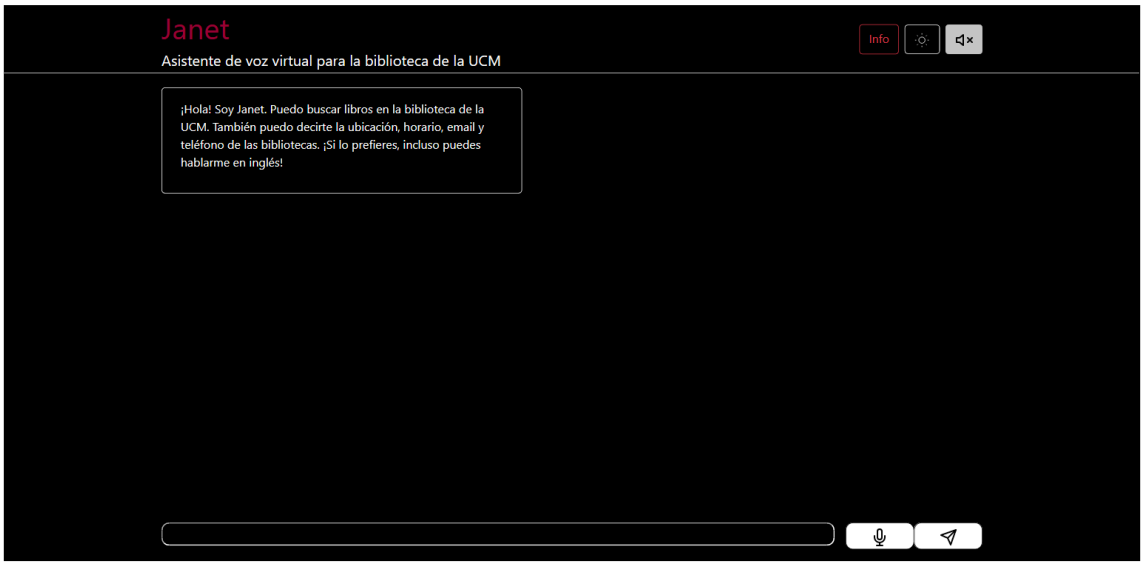

Figura 5.10: Modo de Alto contraste en la Web

se lee el texto de la página correctamente, y con la tecla de tabulador se puede seleccionar los elementos clickables. Sin embargo, se encuentran varios errores en la forma que tenía el narrador de leer estos elementos:

- Varios de los botones no eran accesibles a través de la tecla del tabulador. Esto se debía a un error a la hora de declarar esos elementos ya que no estaban correctamente marcados como botones, añadiendo correctamente estas etiquetas se solucionó el problema.
- La narración de algunos de los elementos no era muy descriptiva: este problema se solucionó empleando el atributo *aria-label* que prevalece por encima de los demás ante el narrador, de tal manera que siempre lo lee.
- Los enlaces a múltiples libros generados mediante JQuery no eran descriptivos: los enlaces tenían todos el mismo titulo "Más Información"de manera que era imposible detectar para que libro era que enlace. Este problema se solucionó añadiendo en JQuery en un atributo *aria-label* para estos elementos, incluyendo en su interior el título del libro devuelto de la consulta.

5.6. Diseño de clientes móviles

Tras los cambios hechos al bot, se ha visto necesaria la modificación de las interfaces de los móviles. Las razones son las siguientes:

- La dirección en la que se encuentra hospedado el servidor ha cambiado. Esto significa que los clientes móviles antiguos envían peticiones a una dirección en la que ya no va a recibir ninguna respuesta. Aunque esto se soluciona temporalmente con una redirección de la dirección antigua a la nueva, es preferible evitar este paso intermedio.
- Pretendemos que todas las interfaces se asemejen, obteniendo así una coherencia visual mayor. Las aplicaciones móviles cumplían su función. No obstante, se cree que la foto de la UCM en el fondo obstruía la claridad visual, y la disparidad entre

versiones hace que no parezca una única aplicación con la misma funcionalidad. Para solucionarlo, se quiere orientar el diseño a uno más simple y con los colores de la universidad.

- Con la web, se ha introducido la posibilidad de introducir las consultas como texto. Se pretende que esta funcionalidad también llegue a los clientes móviles.
- Se quiere introducir el modo de alto contraste en la versión de Android, a la vez que mejorar la ya existente en iOS.
- La función de texto a voz requiere un idioma. Cuando la aplicación antigua lee texto en inglés con la voz española, el resultado es una pronunciación incomprensible. Para ello, necesitamos que la aplicación detecte cuándo el texto es en inglés, y que se cambie la voz de manera adecuada para adaptarse a ello.

Los cambios vienen motivados por estas razones y a su vez por algunas imperfecciones que se han podido observar en estas. A continuación, se detallarán los detalles de los cambios realizados en cada plataformas.

5.6.1. Android

Android es la versión del cliente que más se ha modificado, tanto visualmente como de forma funcional. *Janet* fue diseñada con la versión de iOS como el principal objetivo. Es por esto por lo que se pueden percibir varios aspectos que no estában pulidos de forma suficiente.

5.6.1.1. Funcionalidad

Una funcionalidad notable que desaparecía frente a la versión de *Janet* para iOS es el modo de alto contraste. Este modo utiliza solo el color blanco y negro para mostrar la información, facilitando así la lectura. Ya que es una aplicación orientada a la accesibilidad, se cree necesario que exista este modo para ayudar a las personas con problemas de visión. Esto se hace en iOS desde la configuración fuera de la app, pero dado que esto es más difícil de hacer en Android, se decide integrar en la propia aplicación.

Para esto se agrega un botón de opciones en la barra superior. Desde ahí se accede a otra pantalla en la que se puede escoger si se activa o no el modo de alto contraste. Si se activa se cambia inmediatamente la visualización, y este cambio se mantiene para futuras sesiones en la aplicación.

En esta pantalla de opciones se agrega también la otra opción que faltaba respecto a la aplicación de iOS. Dicha opción permite seleccionar si queremos que la app lea los mensajes entrantes en voz alta o no. Por defecto, está activada, para ayudar a los usuarios con discapacidad auditiva y dotar de una mayor personalidad al bot. Esta opción también permanece entre ejecuciones distintas de la aplicación.

Otra funcionalidad que se agrega es la posibilidad de introducir los mensajes en un campo de texto en lugar de enviarlos por voz. Esta se implementa en forma de un campo de texto junto a un botón en la parte inferior de la pantalla, que hace la misma función que la entrada por voz sin tener que transcribir el audio.

Para poder soportar los correos de las bibliotecas, también se maneja la recepción de este tipo de mensaje. Al pulsar sobre la dirección de correos, se solicita la apertura de un cliente de correo instalado en el teléfono.

Existe otro aspecto diferente que la aplicación de Android no poseía pero la de iOS sí. Esto es, al obtener más de un libro, se puede pulsar en uno de ellos para obtener información más completa del mismo, como puede ser la disponibilidad y el enlace a la web del mismo. Se ha hecho que esta funcionalidad también este disponible de forma idéntica en esta versión.

Por último, se añade la voz en inglés para los mensajes en inglés. Los mensajes recibidos del servidor de *Janet* vienen con un campo en el que se especifica el idioma del mensaje. En función de este campo, la aplicación cambia la configuración del idioma de la voz para adaptarse al mensaje.

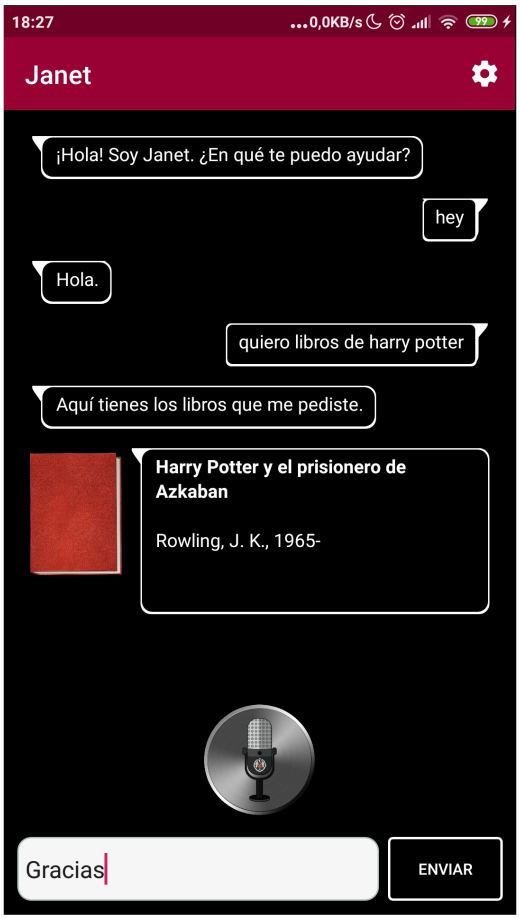

Figura 5.11: Modo de alto contraste en la aplicación de Android

5.6.1.2. Aspecto visual

La app era notablemente diferente a la versión de iOS. A diferencia de esta, se utilizaban diferentes colores para representar los mensajes entrantes y salientes, y carecía de cualquier tipo de animaciones.

Los globos en los que se introducían los mensajes del chat se generaban mediante un **mapa de bits de tamaño modificable (o imagen "9-patch")**. Esto es una imagen, la cual se divide en nueve fragmentos. Estos fragmentos son las cuatro esquinas de la imagen, cuatro lados, y la parte central. Con esa división, se puede ampliar cada lado de forma indefinida sin estirar las imágenes ni causar una pérdida en su resolución, como podemos ver en la Figura 5.12. El problema de esta imagen era que estaba generada con una resolución muy pequeña, y en los bordes se podían percibir los píxeles por los cuales la imagen estaba formada.

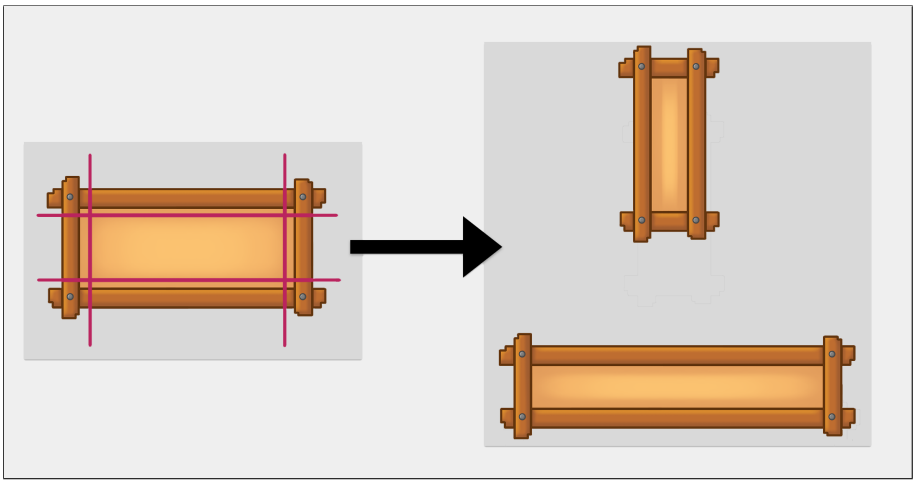

Figura 5.12: Ejemplo de imagen "9-patch", la cual puede deformarse sin perder calidad

Para mejorar esto, se crea un "drawable". Esto es cualquier tipo de gráfico que puede ser dibujado en pantalla. La imagen que se utilizaba anteriormente también es un "drawable". Sin embargo, esta vez se utiliza código en XML para definir cómo deben dibujarse los globos en lugar de una imagen. De esta forma es más fácil adaptarse a cualquier tamaño de pantalla, y desaparecen los problemas de la baja resolución en los bordes. Adicionalmente, se puede cambiar el color de cada globo fácilmente desde el código en lugar de tener que generar una imagen nueva.

Otro aspecto diferente a la versión de iOS era cómo se manejaba la interfaz tras mandar un mensaje. Se ha hecho que al enviar un mensaje, desaparezca tanto el cuadro de texto, el botón para hablar, y el botón de enviar mensaje, y que en su lugar aparezca un símbolo de carga. Así se muestra fácilmente cómo se está esperando una respuesta, y se evita que el usuario intente enviar mensajes hasta que se reciba.

5.6.2. iOS

La aplicación para iOS era de la más completa de entre las dos. Tenía opciones para establecer el modo de alto contraste y silenciar la lectura del texto desde fuera la aplicación. Además estaba completa con animaciones para cada acción. A pesar de su completitud, se altera para que tenga un aspecto visual coherente con el resto de aplicaciones, y añadimos funcionalidades.

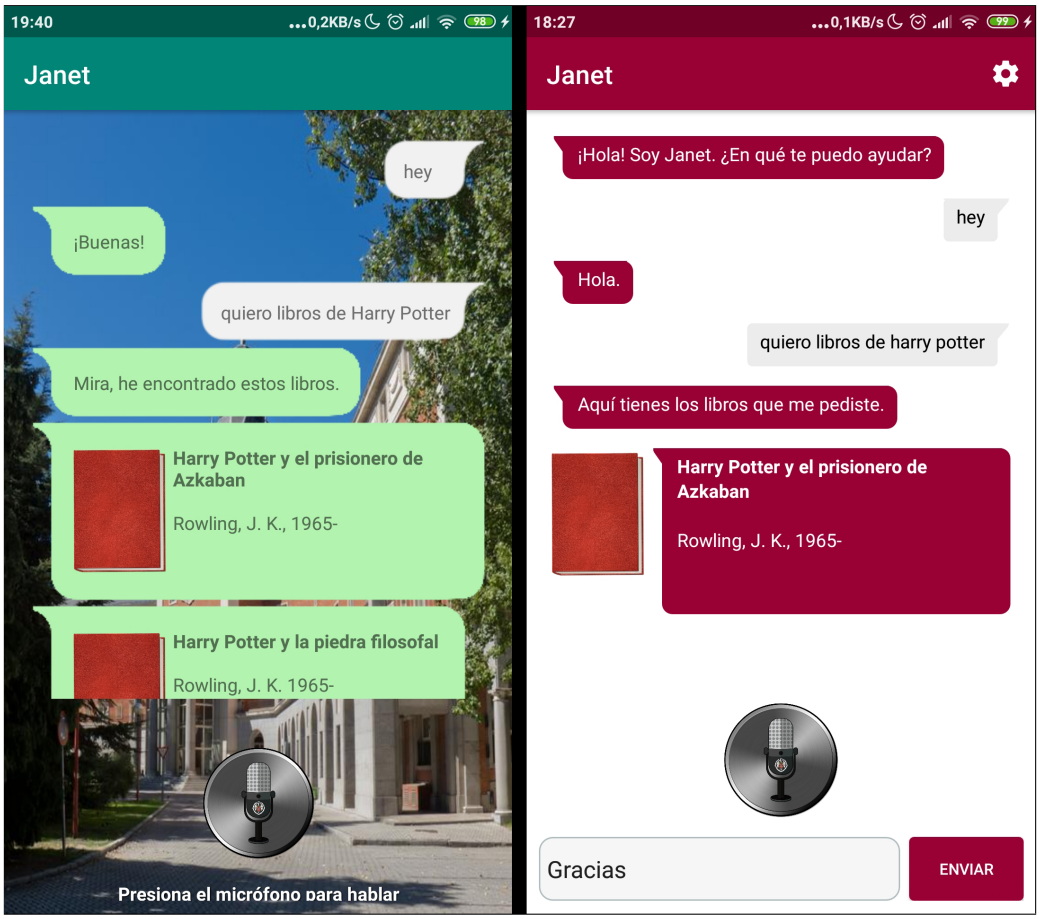

Figura 5.13: Comparación visual de la antigua frente a la nueva versión del cliente Android

5.6.2.1. Funcionalidad

La principal función añadida, al igual que para la versión Android, es la posibilidad de enviar mensajes mediante un campo de texto. Se coloca en la parte inferior de la aplicación, manteniendo el tamaño del botón de voz, para que este pueda seguir siendo fácilmente accesible.

Al recibir mensajes con un correo, agregamos la opción de que el usuario pulse la dirección para escribir un correo. También se soluciona un bug, en el que no se abría la agenda al pulsar sobre un número de teléfono como se pretendía.

Aunque el modo de alto contraste existía, no funcionaba de forma correcta. Los primeros mensajes que aparecían en la aplicación seguían teniendo el aspecto normal, y solo cambiaban al modo de alto contraste cuando el usuario se desplazaba arriba o abajo por el conjunto de mensajes. Tras arreglar este problema causado por el orden erróneo del código, agregamos también código para que la entrada de texto agregada también cambie con este modo.

Para solucionar el problema de la voz en inglés, inicializamos en dos variables distintas dos voces, una en inglés y otra en español. La voz que se utiliza para hablar al usuario se selecciona antes de leer, tras inspeccionar el lenguaje del mensaje respuesta.

Figura 5.14: Modo de alto contraste en la aplicación de iOS

5.6.2.2. Aspecto visual

La principal reforma del aspecto visual se hace para adaptarse al esquema de colores de la web. Al igual que en la versión Android, se elimina la imagen de fondo del decanato, sustituyéndola por un fondo blanco.

Los globos de los mensajes de texto también se formaban mediante imágenes. A pesar de ello, estas imágenes tenían una alta resolución, por lo que se pueden mantener para esta versión. El único cambio realizado a los globos es un cambio de color, estableciendo el color gris para mensajes salientes y el color cardenal típico de la UCM para los entrantes.

5.6.3. Compatibilidad con versiones anteriores

Un tema que resulta de especial importancia es la compatibilidad de las versiones anteriores de las aplicaciones con las nuevas. Aunque no se pretende que se vean limitadas las nuevas funcionalidades por hacer que estas versiones sigan funcionando, se desea que al menos la funcionalidad básica siga intacta.

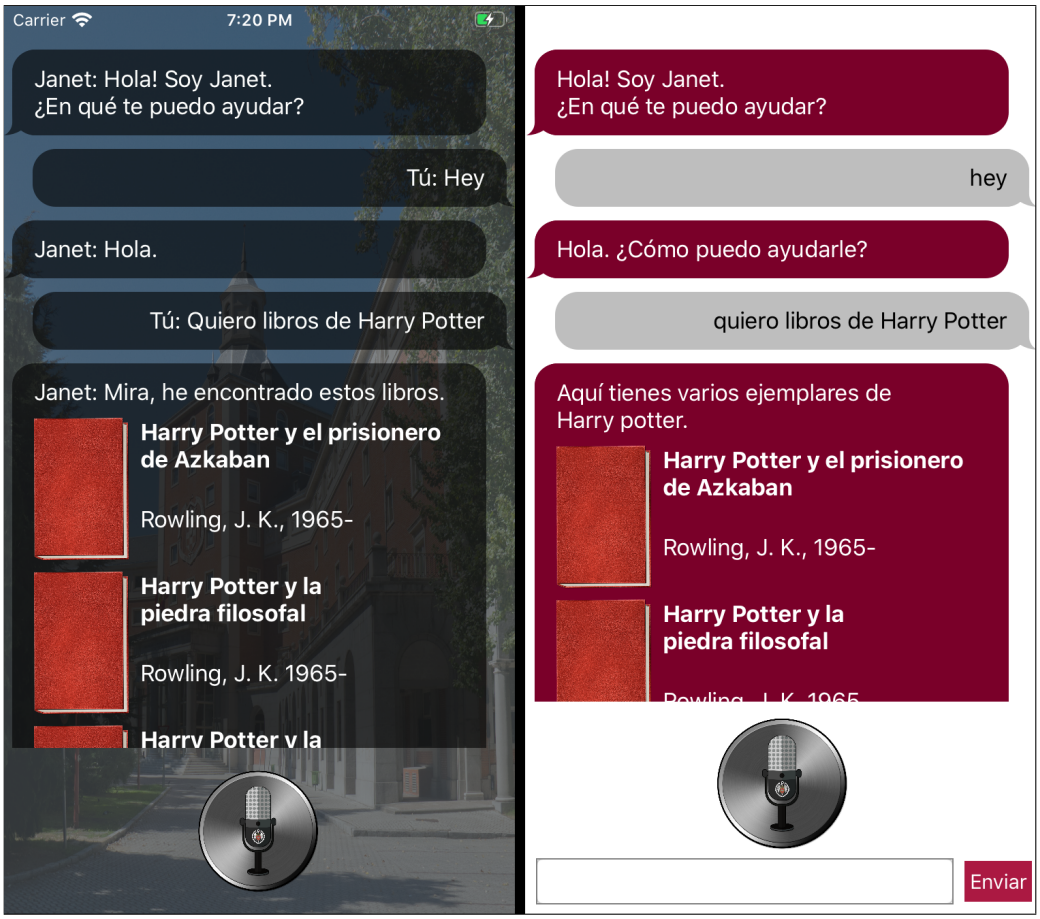

Figura 5.15: Comparación visual de la antigua frente a la nueva versión del cliente iOS

Cambios como agregar la entrada mediante campos de texto, o el modo alto contraste, son solo funcionalidad añadida. Es decir, no están disponibles en las versiones antiguas, pero no impiden su funcionamiento. Sin embargo, hay varias funcionalidades cuyo funcionamiento en versiones antiguas se debe tener en cuenta:

- Las versiones antiguas de la aplicación pueden comunicarse en inglés con *Janet* también, ya que la gestión de esto se hace en el servidor. Desafortunadamente, la voz de las aplicaciones están configuradas a español. Esto quiere decir que cuando se recibe un texto en inglés, la lectura se hace en pronunciación española, causando un diálogo difícil de comprender.

- Los mapas de localizaciones no funcionan en versiones antiguas de la aplicación de Android. Esto se debe a que utiliza un token de la API de Google Maps caducado, el cual ha sido recientemente sustituido por uno funcional. Se siguen dando las calles de la localización de bibliotecas, pero no se muestra un mapa ni se puede obtener un enlace a Google Maps de forma directa.

- El nuevo tipo de consulta, la petición de correos electrónicos, no funciona para las versiones antiguas. Al ser un nuevo tipo de mensaje, la gestión de este no está preparada. No obstante, las aplicaciones toman la acción por defecto, y tratan el mensaje recibido como otro cualquiera, sin mostrar la dirección de correo.

A pesar de que se pierdan estas mejoras en versiones antiguas, se pueden considerar pérdidas asumibles que en ningún caso causan fallos en la aplicación. Hay un último problema sobre el que hay que llamar la atención: El identificador de los usuarios.

Como se ha comentado en la sección 5.4, la forma en la que se generan en la web los identificadores de usuarios es mediante caracteres generados aleatoriamente. Esto causa un problema a la hora de asignar nuevos identificadores en las versiones móviles.

Anteriormente, los identificadores se asignaban con un entero secuencial. Es decir, el primer usuario tendría como identificador "1", el siguiente "2", etc. Estos se trataban como un número entero, y cuando se quería asignar a un nuevo usuario, se tomaba el último y se le sumaba 1. Como los identificadores tanto de la web como de las aplicaciones se almacenan en la base de datos, si se intenta asignar un ID de móvil de la manera descrita después de que el último sea un ID web, al intentar sumar 1 a una cadena de caracteres, ocurre una excepción por la incompatibilidad de los tipos. Esto hace a las aplicaciones móviles sin identificador ya asignado inservibles.

Afortunadamente, la solución es controlable sin tener que cambiar el código de las aplicaciones móviles, ya que quien asigna la identificación en estas es el servidor. Consiste en asignar los identificadores de la misma forma que en la web, sin tener que utilizar enteros secuenciales. Generando de forma aleatoria cadenas de 130 caracteres, se simplifica la creación de identificadores y se soluciona el problema de incompatibilidad.

5.7. Instalador

Se han introducido múltiples mejoras al instalador. Anteriormente, el instalador movía la aplicación a una ruta fija, la carpeta del usuario tfg-biblio, el cual también era creado durante la instalación. Por lo tanto, esta instalación fallaba si ya existía dicho usuario o la carpeta.

Para remediarlo, se ha introducido al script la opción de que sea el propio usuario quien especifique dónde se realiza la instalación y con qué usuario. La ruta se intenta crear si no existe, igual que el usuario. Si ya existe, la instalación continuará. Además, para informar de esta funcionalidad, al instalador se le proporciona una breve sección de ayuda.

Estos parámetros de la instalación se introducen de forma sencilla con una opción en la línea de comandos, para que el esfuerzo que tenga que realizar el usuario durante la instalación sea mínimo. Además, por defecto, utilizará el usuario tfg-biblio y lo instalará en la carpeta dentro de home con su mismo nombre.

A su vez, se ha implementado la creación de un entorno virtual, para que dentro de este se puedan instalar las diferentes dependencias de Python en las versiones requeridas. De esta forma, se evitan posibles conflictos que se podrían dar con las dependencias instaladas en el sistema operativo.

5.7.1. Actualizaciones

Uno de los problemas del instalador original era que este falla si se ejecuta sobre una máquina en la que ya se ha instalado *Janet*. Si la carpeta donde se intenta instalar ya está ocupada por una instalación previa, el proceso no se puede llevar a cabo.

Para remediarlo y poder actualizar *Janet* con las nuevas versiones generadas, creamos una función de actualización dentro del instalador. Si el instalador detecta que en la carpeta designada ya contiene los archivos de *Janet*, permitirá actualizarla. De esta manera, podemos hacer ambas funciones con un único

La actualización consiste en borrar las carpetas antiguas de *Janet* e introducir las nuevas. Sin embargo, no trata de crear un usuario, los servicios, ni hace la instalación de nuevo del entorno virtual. Al igual que el anterior instalador va a requerir las claves de la API de WolrdCat para ser ejecutado, las cuales no se proporcionan en el repositorio.

Con este script además se hace una nueva instalación de dependencias de Python en el entorno virtual, con lo cual podemos instalar nuevas versiones de paquetes o nuevas dependencias sin tener que reinstalar cada paquete. Lo vamos utilizando durante el desarrollo para probar las nuevas versiones, y poder rápida y cómodamente descargarnos e instalarnos el nuevo código sin tener que generar una máquina virtual limpia.

Por último, también se eliminará el servicio de Jarvis si se encuentra creado en el sistema. Este servicio, como se ha comentado en el punto 5.3, ha dejado de ser útil. Es por esto por lo que mantenerlo en el sistema sería completamente innecesario además de no funcionar, por lo que a parte de no instalarse en futuras versiones, también se borra en caso de actualización.

5.8. Problemas con la arquitectura del servidor

Surgen varios problemas nacidos de la arquitectura dada para hospedar el proyecto. Ya que no se tiene un control directo sobre la arquitectura, se tendrá que aplicar soluciones a los problemas que la estructura genera.

Los dos problemas notables que serán detallados con mayor profundidad en las siguientes secciones son:

- Hay un servidor intermedio entre el que contiene la aplicación y los clientes.
- Solo se dispone de un único puerto para servir aplicaciones al exterior.

5.8.1. Proxy inverso

Esta web, junto a todos los servicios de *Janet*, es servida desde `https://holstein.fdi.ucm.es/tfg-biblio2/`. Este servidor de la UCM en el que se encuentra alojada la aplicación se encuentra tras un proxy inverso (Valeur et al., 2006). Un proxy inverso es un servidor que actúa como intermediario entre los clientes y uno o varios servidores. Este proxy actúa de manera transparente para el usuario, y se utiliza principalmente para filtrar posible tráfico dañino antes de enviar a los servidores la petición del cliente.

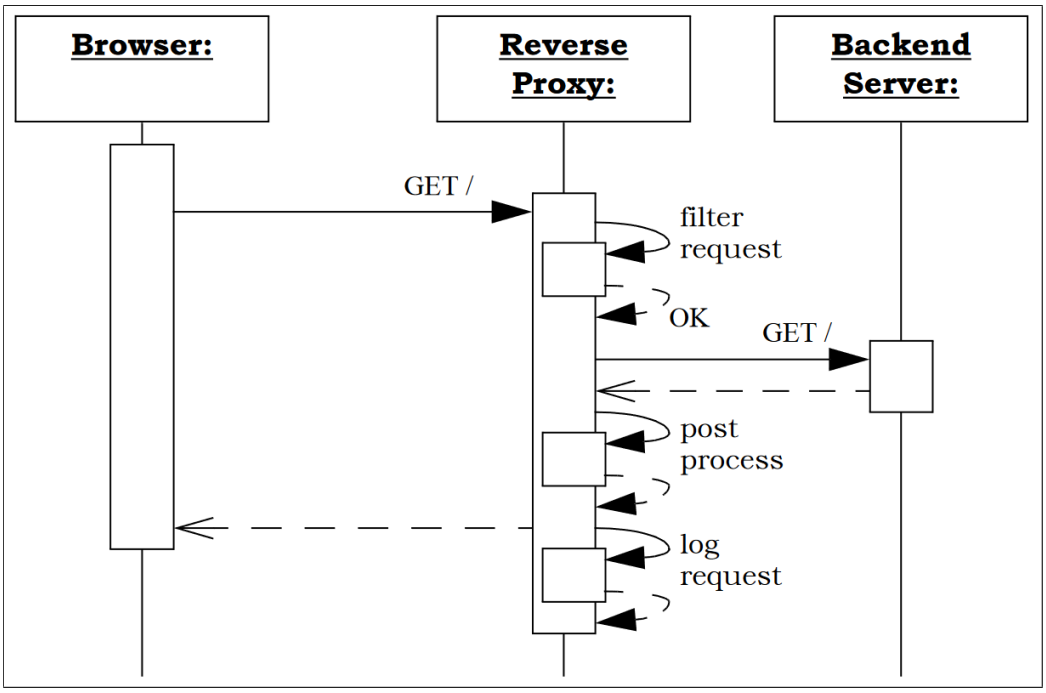

Figura 5.16: Esquema del funcionamiento de un proxy inverso

La conexión con el servidor sigue los siguientes pasos, de forma específica a nuestra aplicación:

1. El cliente pide una página a `https://holstein.fdi.ucm.es/tfg-biblio2/`.
2. El proxy inverso, localizado en `https://holstein.fdi.ucm.es` recibe la petición.
3. El proxy observa que se ha pedido la URL "tfg-biblio2", por lo que este le reenvía la petición al servidor que se encarga de manejar ese proyecto.
4. El servidor de "tfg-biblio2" procesa los datos, y devuelve una respuesta al proxy.
5. El proxy devuelve al usuario la respuesta recibida por el servidor

Flask no puede saber por si mismo que se encuentra tras un proxy inverso. Al ser transparente, Flask actúa como si la raíz del proyecto se encontrase en `https://holstein.fdi.ucm.es`, ignorando la parte de "tfg-biblio2". Si quisiera importar una hoja de estilos que se encuentra en "static/css/style.css", tendríamos que acceder desde la URL completa `https://holstein.fdi.ucm.es/tfg-biblio2/static/css/style.css`. Sin embargo, Flask intenta acceder a `https://holstein.fdi.ucm.es/static/css/style.css`. Estos accesos incorrectos causan errores 404, impiden que se carguen las hojas de estilos, los ficheros de Javascript... En general, sin esta funcionalidad, nuestra web es incapaz de funcionar ni visualizarse de forma correcta.

Para solucionar este problema, utilizaremos Flask Reverse Proxy Middleware[5]. Esta sencilla librería introduce un middleware a Flask que agregará el prefijo deseado a las URL que se quieran direccionar. Configurándola para que agregue "tfg-biblio2" a cada petición,

[5](`https://pypi.org/project/flask-reverse-proxy-fix/`)

se consigue que detecte tanto los ficheros de Javascript como las hojas de estilo. Sin embargo este fragmento de código solo es necesario si se utiliza la web desde un proxy inverso, y debería ser eliminado si se sirviese directamente.

Otro de los problemas que resultaron de la implementación en un servidor fue que las peticiones *URL* desde los archivos de *JQuery* resultaban en errores 404. Tras investigar sobre la causa de este problema se descubre que se debía a que, al igual que el problema anterior, la aplicación trataba de acceder a `https://holstein.fdi.ucm.es` en vez de a a `https://holstein.fdi.ucm.es/tfg-biblio2`. Para solucionar este fallo hicimos uso de la información provista en la página de la documentación de Flask, `https://flask.palletsprojects.com/en/1.1.x/patterns/jquery/`, donde se habla en detalle sobre cómo crear una variable global dentro de los archivos HTML denominada SCRIPT_ROOT. Dicha variable se añadirá al principio de cada petición *URL* para que así accedan a la ubicación en donde está alojada la aplicación, siendo en este caso `https://holstein.fdi.ucm.es/tfg-biblio2`.

5.8.2. Gestión de múltiples servidores

El cliente tanto Android como iOS se conecta mediante HTTP con una URL dada, utilizando el puerto 80. Actualmente, es el mismo servidor el que sirve tanto *Janet* como el cliente web.

Uno de los problemas que se producen en el servidor, es que únicamente disponemos del puerto 80. Esto hace que *Janet* y el cliente Web, siendo dos aplicaciones diferentes, no puedan convivir utilizando el mismo puerto.

La solución implementada a este problema resulta sencilla. Se ejecuta el cliente web en el puerto 80, para que pueda ser accesible desde la página principal. A su vez, se emplea una URL dentro de la web, en la que si se entra, el cliente web redirige la petición al puerto en el que esté la aplicación de *Janet*. De esta manera, es posible acceder a ambas desde un único puerto, utilizando distintas direcciones. Además, esta redirección es transparente para el usuario, el cual puede acceder a la API de *Janet* desde la misma URL.

Capítulo 6

Ejemplo de Uso

6.1. Petición de un libro

En este capítulo, utilizaremos uno de los casos de uso más complejos de la herramienta, para mostrar de la forma más completa posible el funcionamiento e interacción de todas las partes del aplicativo.

Al entrar en la **aplicación de Android**, el usuario se encuentra con un mensaje de *Janet*, como se ve en la Figura 6.1, el cual describe la funcionalidad de la misma. Este mensaje viene directamente en la aplicación, por lo que siempre va a estar disponible y no necesita comunicarse con el servidor.

El usuario entonces escribe en el campo de texto la consulta que quiere realizar, que este caso es: "Quiero el libro titulado el Quijote de Cervantes". Una vez se presiona enviar, el texto se envía al cliente web. Cabe destacar que utilizando la entrada de voz, esta se transcribe a texto y a partir de ese punto, se procede de la misma manera sin tener que presionar el botón de enviar. Justo después de enviarse, se desactiva la posibilidad de enviar otro, y se muestra un "spinner" hasta que se obtenga respuesta, como se ve en la Figura 6.2.

El cliente web recibe la petición. En el único puerto disponible reside el cliente web, por lo que al recibir la petición de un cliente móvil, redirige esta al puerto en el que se encuentra el servidor de *Janet*.

El servidor de *Janet* observa la petición. Esta incluye el mensaje, el tipo de petición, y un identificador de usuario. Si este identificador tiene un valor, se utilizará para cargar datos de conversaciones pasadas. Si no, se le asignará a un identificador nuevo. En este caso este usuario ya había accedido anteriormente, por lo que *Janet* puede cargar el tracker e historial de la conversación.

Janet **detecta automáticamente el idioma del mensaje**. Si no se logra identificar como inglés o español, la petición se tomará por defecto en español. En este caso, se identifica con facilidad que el mensaje está en español. *Janet* utiliza el modelo de Rasa en este idioma para averiguar qué intención tiene el usuario y qué acción debe realizar. Además, Rasa también proporciona un mensaje que devolverá al usuario.

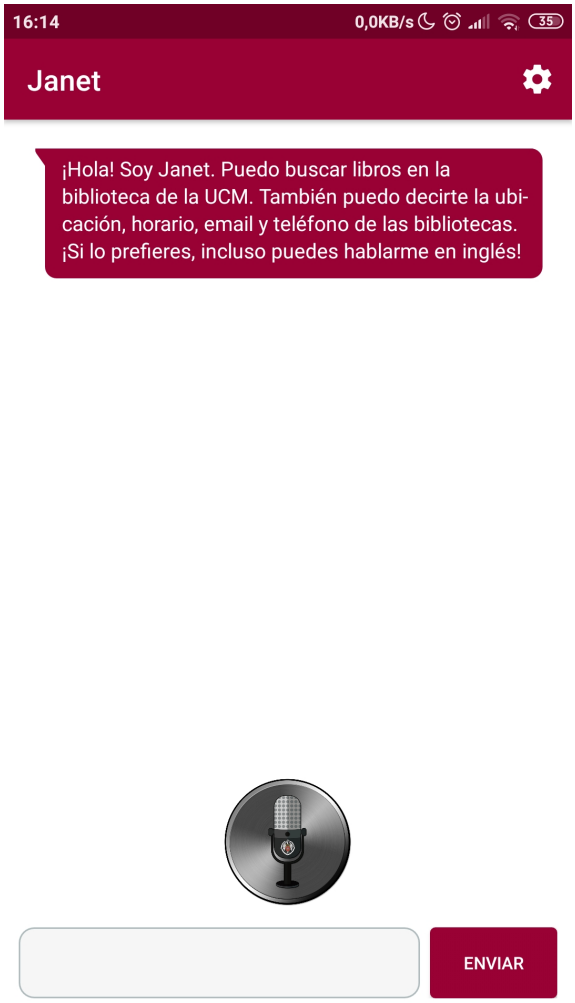

Figura 6.1: Pantalla de la aplicación de Android al entrar en ella

Gracias al modelo de Rasa, el servidor de *Janet* detecta que la intención del mensaje es pedir un libro por título y autor. Como es una acción compleja que requiere varias entidades, Rasa envía la petición al **servidor de acciones de Jarvis**. Este extrae las entidades a la vez y las agrega al tracker, a la vez que devuelve un respuesta de texto.

Se han extraído las entidades del mensaje y las ha agregado al tracker. Estas entidades son "El Quijote" como título, y "Cervantes" como autor. *Janet* entonces utiliza estas entidades y la intención para realizar la acción correspondiente, la cual es enviar la petición a la **API de WorldCat** para obtener la información.

La obtención de la información del libro se hace en **tres pasos** utilizando la API:

1. Se obtiene del libro el título, el autor, el ISBN y el código OCLC del libro.
2. Se utiliza otra llamada con el código OCLC para obtener la url al libro.
3. Una tercera llamada es necesaria para comprobar la disponibilidad del libro dentro de las bibliotecas de la UCM utilizando el código OCLC.

Ya se ha obtenido toda la información necesaria y se puede devolver al cliente web, el

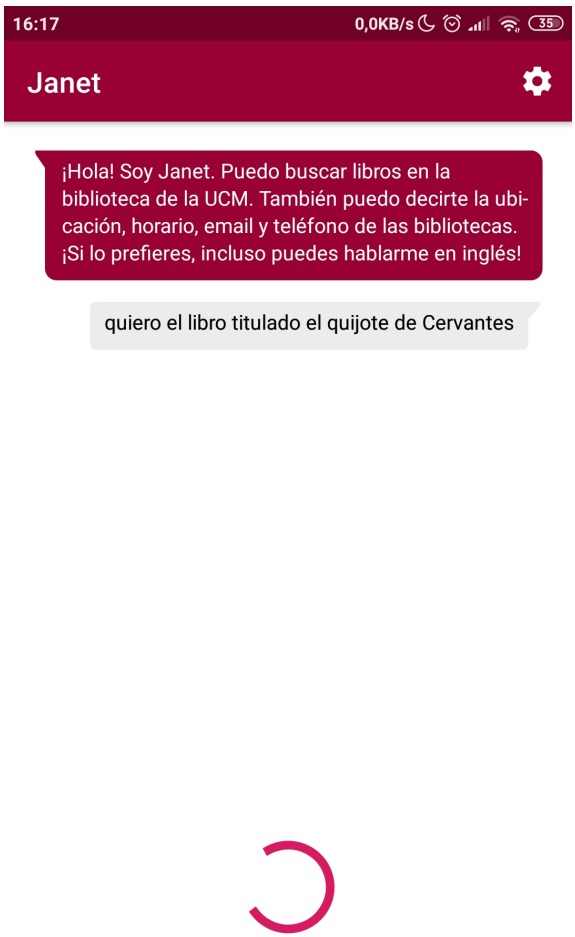

Figura 6.2: Pantalla de la aplicación mientras se recibe la respuesta

cual se lo envía al cliente Android. Los campos que se devuelven son el identificador del usuario, el mensaje, un tipo de respuesta que será "libro único", la información del libro, el idioma de la respuesta, y un código de error, el cual es 0.

El cliente móvil utiliza toda esa información para mostrársela al usuario. Se muestran los globos de texto, se lee el texto al usuario, y se muestran los datos del libro encontrado. Además, se vuelve a reactivar la entrada de texto y voz.

La única parte que falta es la imagen, que no se obtiene mediante la API de WorldCat. El cliente Android hace una llamada en paralelo a la API de OpenLibrary para conseguir la imagen, y la actualiza en cuanto la tiene, quedando como la Figura 6.3.

6.2. Peculiaridades

Aunque el caso de uso anterior describe de forma general el funcionamiento de la aplicación. Sin embargo, existen un par de peculiaridades que resultan interesantes de comentar.

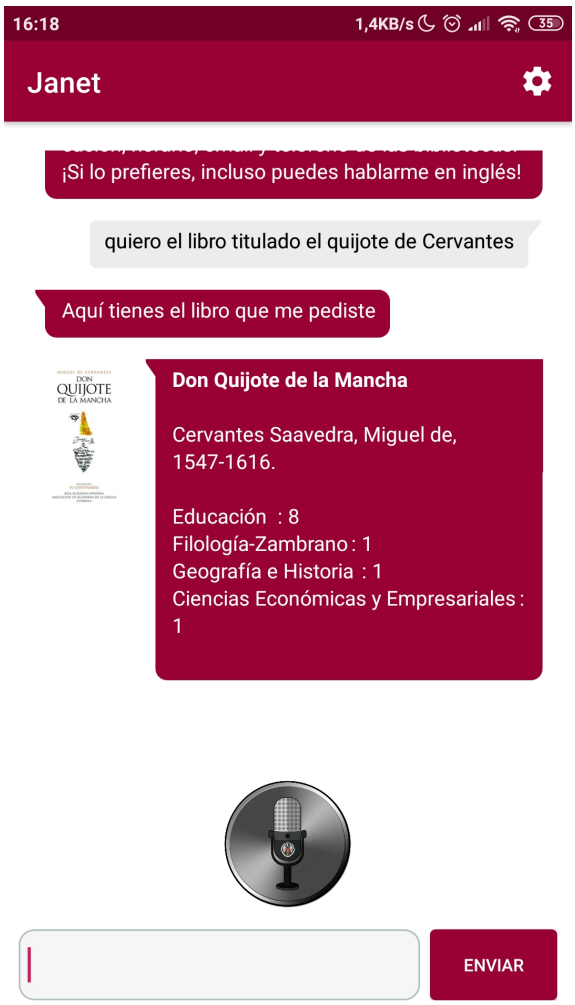

Figura 6.3: Resultado final tras obtener respuesta del servidor

- Si el mensaje se hubiera intentado mandar con un mensaje de voz por el cliente web, el mensaje es tratado de la siguiente manera:

  1. La voz se graba en la parte del cliente y se envía el clip de audio al servidor web.
  2. Este clip de voz se transcribe en el servidor, y se devuelve la transcripción la usuario.
  3. A partir de este punto, el flujo es igual que al descrito en el ejemplo, enviando el texto transcrito.

- **Existe otro tipo de petición**, en el que no se tiene que utilizar las habilidades conversacionales de Rasa. Para este tipo de consultas, denominadas "OCLC", el cliente le pide la información detallada de un libro. Estas se hacen para obtener la información detallada de un documento tras haberlo obtenido de una lista.

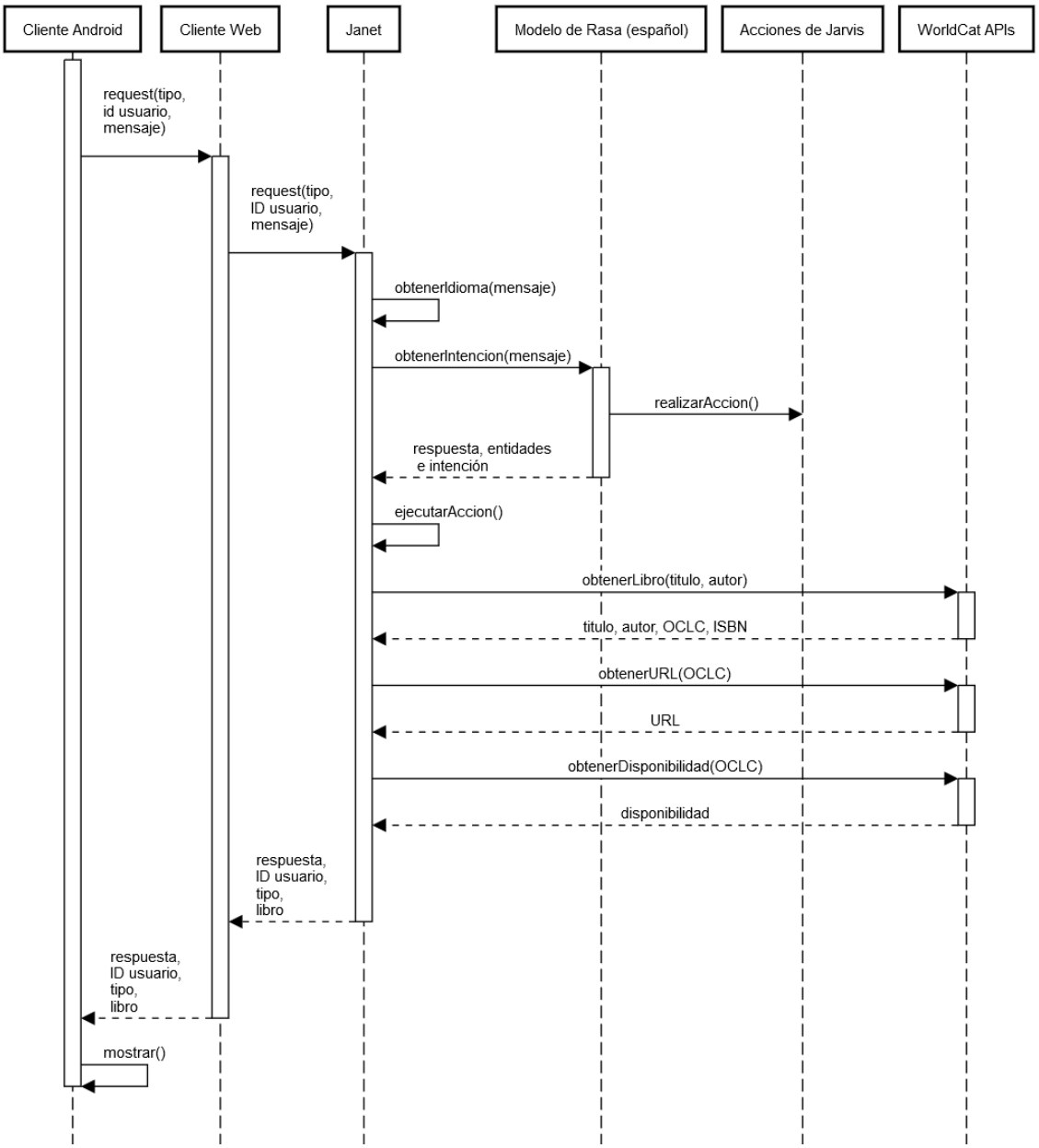

Figura 6.4: Diagrama de secuencia del ejemplo del libro

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

Inspirados por el trabajo realizado anteriormente por nuestros compañeros, nos unimos a este trabajo con el objetivo de mejorar la aplicación y pudiese ser utilizable en un futuro. Para ello, nos planteamos diversos objetivos que pudiesen hacer de ésta, algo más completo que cualquiera pudiese utilizar.

Hemos creado una página web utilizando las tecnologías de Flask en Python y JQuery con Javascript. Con esta, el acceso a la aplicación es inmediato, utilizando únicamente el navegador. Se ha conseguido, pese a las dificultades por los tipos de navegador y los permisos de usuario, trasladar toda la funcionalidad de las aplicaciones móviles a este nuevo cliente.

La mayor mejora que se ha hecho ha sido en el funcionamiento del interpretador del lenguaje. La actualización de Rasa, la cual estaba en fase beta cuando se inició este proyecto, ha supuesto un desafío a la hora de reorganizar la arquitectura. El chatbot consigue interpretar mejor los mensajes del usuario, ya no solo por esta actualización, sino también por el incremento en los casos de entrenamiento, mejora de las historias, e incremento de los tipos de intenciones disponibles.

Otro de los hitos cumplidos ha sido la posibilidad de poder utilizar a *Janet* en inglés. Se ha querido además que la interpretación del lenguaje sea hecha a través de cada uno de los mensajes, de forma que el usuario nunca tenga que especificar en qué idioma está hablando. Finalmente se ha logrado este objetivo, superando las dificultades que supone que el diseño original no contase con la traducción a distintos lenguajes.

A pesar de nuestra inexperiencia en el desarrollo de aplicaciones móviles, también hemos actualizado tanto la versión iOS de la aplicación como la de Android. Motivados por el deseo de una coherencia tanto funcional como visual entre todas las versiones de la aplicación, se han cambiado completamente para adaptarse a las nuevas funcionalidades, obteniendo a su vez un aspecto visual más rico. Esto es especialmente cierto en la aplicación para Android, para la se han implementado múltiples funcionalidades que se perdían desde la versión de iOS.

Gran parte de nuestro esfuerzo y tiempo ha ido, de forma inesperada, a solucionar

problemas técnicos derivados de la estructura Debian, los instaladores, y la actualización de Rasa. Creemos haber logrado una forma más sencilla y compatible de instalar y utilizar todo el aplicativo, de forma que pueda ser más fácilmente mantenible y ampliable para quien quisiese trabajar en un futuro con esta aplicación.

Pese a todo, también hay puntos negativos. Entre los objetivos que no hemos podido cumplir, el más notable ha sido no poder utilizar las entradas de los usuarios para mejorar el comportamiento esperado de la aplicación. A pesar de haber utilizado el feedback de amigos y familiares, la aplicación no ha podido ser empleada por usuarios reales, tanto por los problemas dadas en la obtención y posterior ejecución del servidor, como por la desorganización de nuestra parte. Además, aunque era un objetivo secundario, tampoco hemos podido ampliar Janet a otros servicios ofrecidos por la UCM ajenos a la biblioteca.

Como conclusiones finales, podemos decir que el proyecto se ha desarrollado con éxito, cumpliendo la mayoría de objetivos planteados en su inicio. Se ha conseguido incrementar la funcionalidad de la aplicación, y se ha aumentado su accesibilidad gracias a la creación de la versión Web y a la actualización de los clientes móviles. Con las mejoras de este asistente, creemos que se ha convertido una buena base para que pueda ser utilizado por la universidad, y esperamos que sirva para facilitar el acceso a los servicios ofrecidos por la biblioteca.

7.2. Líneas de trabajo futuro

El proyecto sigue requiriendo algunas de las posibles mejoras que no se han implementado respecto al año anterior. Junto a ellas, podemos incluir otras mejoras que por falta de tiempo o por ser demasiado grandes en alcance, no hemos podido implementar, pero sería interesante plantear de cara a un futuro.

- **Los modelos de entrenamiento pueden seguir mejorándose:** Incrementar el número de casos de entrenamiento siempre ayudará a mejorar el comportamiento del bot. Además, también se pueden agregar más tipos de intenciones o diferentes historias.
- **Utilizar la información recopilada de usuarios reales:** Si estas aplicaciones llegan a tener un uso real y serio, se podría utilizar la información de conversaciones para editar los modelos de entrenamiento de una forma mas precisa. Los mecanismos de recogida de información ya existen en la aplicación, y se guardan en la base de datos de MongoDB. Se puede por ejemplo utilizar métodos de análisis de datos para observar las conversaciones y reconocer intents que el usuario espera y no existen en la actualidad.
- **Ampliación de los servicios de biblioteca de *Janet*:** Algunas de las posibles funcionalidades interesantes que podría dar este servicio serían la reserva de libros tras buscarlos, poder ver la disponibilidad de salas de trabajo en equipo, o reservas de las mismas.
- **Ampliación de *Janet* a otros servicios ajenos a la biblioteca:** Aunque la aplicación naciese como un asistente virtual para la biblioteca de la UCM, podría ampliarse a otros campos que resulten útiles. Estos podrían ser, por ejemplo, información sobre las prácticas en empresa, o incluso el acceso a datos del campus virtual.

- **Integración con la cuenta UCM:** Esta integración podría permitir, entre otras cosas, ver el estado de los préstamos de libros. Además, este tipo de integración sería necesaria si se quiere ampliar a otros servicios, como los que se han mencionado anteriormente.
- **Soporte para otros idiomas:** Aunque con el inglés y el español quedan bastante cubiertos los idiomas que se utilizan en el entorno universitario, se pueden incorporar otros idiomas que también utilicen los estudiantes. La forma en la que se incorpora un nuevo idioma puede ser compleja, por lo que también se podría refactorizar el código con el propósito de encontrar una forma más sencilla para añadir nuevos lenguajes.
- **Información más completa sobre los recursos de la biblioteca:** Se puede mostrar más información sobre los libros que se buscan en la aplicación. Entre estos, se podría mostrar el tipo de recurso (si es una película, libro...), información bibliográfica, ISBN...
- **Delegar funcionalidad a las aplicaciones móviles:** Los dispositivos móviles son cada vez más potentes y los modelos de machine learning están cada vez más optimizados en cuanto a espacio y a eficiencia, por lo que no se puede descartar la opción de dotar a las aplicaciones con cierta funcionalidad como bases de datos o la capacidad de responder consultas simples sin necesidad de conexión al servidor.

Chapter 8

Conclusions and Future Work

8.1. Conclusions

Inspired by the work made by the previous students, we joined this project with the intention of improving the application and making it usable in the future. To that end, we planned a series of different objetives so this application could become something more complete that anybody could use.

We have created a webpage using tecnologies such as Flask in Python and JQuery in Javascript. With this web version, the access to the application is inmediate, needing only a web browser to access it. Although we had to confront some difficulties such as different types of browser and the user permissions, we have managed to translate all the functionalities from the mobile Apps into this new client.

The most important improvement we have made has been the one for the language interpreter. The Rasa update, which was in beta phase when this project started, has supposed the biggest challenge to the architecture restructure. The chatbot manages to interpret the user messages better than before, not only because of the upgrade, but also because of the huge increment in the number of training cases, the improvement of the stories, and the increment un the number of available intents.

Another landmark was achieving the posibility of using *Janet* in english. This also was implemented in a way that the user does not need to specify the language of the message, the application detects the language of the message the user sends. We have managed to overcome the struggle of the original design didn't think about having translations to different languages.

In spite of our lack of experience in the develoment of mobile applications, we have upgraded the versions for both clientes, iOS and Android. Motivated by the desire of cohesion, we have also change the applications visually and functionally so that every version of the application has the same functionalities, obtaining a more professional visual aspect. This is specially true for the Android version, which lacked many of the features that the iOS version had.

A huge part of our effort and time has gone into fixing multiple of the tecnical problems

derived from Debian's structure, the installers, and Rasa upgrade. We believed that we have accomplished a more simple and compatible way of installing and using the application, in a way that is also more maintainable and expandable for anyone that wants to work with this project in the future

Despite all the acomplishments, there are also some negative points. The most notable obtective that we haven't managed to accomplish was using the user entries to improve the behaviour of the application. Although we have used the feedback from family and friends, the application hasn't been tested with real users. This was due to multiple factors, from the struggle of having to find a server where we cold run it, to the disorganization between the team members. We have also not been able to expand *Janet* to other UCM services outside from the library.

In conclusion, we could say that our project has been a success, acomplishing most of the objectives planned from the beggining. We have managed to increse the application's functionality, and the accessibility has been increased thanks to the creation of the Web version and the update of the mobile clients. With the improvements done to this assistant, we believe it has become a good foundation so it can be used by the university, and we hope it becomes useful to help access the services offered by the library.

8.2. Future Work

The project still requires some improvements that haven not been implemented in relation to the previous year. Among theme, we can include other improvements that due to lack of time or the fact that they are too large in scope, we have not been able to implement, but it would be interesting to think about it in the future.

- **The training models can still be improved:** Increasing the number of training cases will always improve the behaviour of the bot. Furthermore, more types of intents or different stories can be added.
- **Information collected from real users can be used:** If these applications end up having real usage, the information from conversations can be used to edit training models in a more precise way. Mechanisms to collect that information already exists in the application, and is stored in a MongoDB database. For example, methods of data analysis can be used to observe conversations and recognize intents that the user expects but are nowhere to be found in reality.
- **Expansions of *Janet's* library services:** Some of the possible interesting features that this service could provide would be the book reservation after searching them, being able to see the availability of cooperation rooms, or the reservations of said rooms.
- ***Janet*'s extension to other non-library services:** Although the app was born as a virtual assistant for the UCM library, it could be extended to other useful fields. These could be, for example, providing help about internships, or even access to virtual campus data.
- **Integration with the UCM account:** This integration could allow, among other things, to see the status of book loans. Furthermore, this type of integration would be necessary if you want to expand to other services, such as those mentioned above.

- **Support for other languages:** Although English and Spanish cover most of the languages used in the university environment, other languages used by students can be incorporated. The way in which a new language is incorporated can be complex, so the code could also be refactored in order to find an easier way to add new languages.
- **More complete information about library resources:** More information about the books that are searched for in the application can be displayed. Among these, the type of resource (whether it is a movie, book...), bibliographic information, ISBN...
- **Delegate functionality to mobile applications:** Mobile devices are increasingly powerful and machine learning models get smaller and faster with time, so it could be an option to give mobile apps the some functionality such as data bases or the ability to answer simple queries without connecting to the server.

Capítulo 9

Aportaciones individuales al proyecto

9.1. Miguel Ángel Castillo Moreno

Para este trabajo he participado en cada parte del desarrollo. Me he implicado en los comienzos de la página web, organizando la estructura de las páginas. También he creado la página de privacidad, he manejado la generación de identificadores de usuarios, la conexión con el servidor de *Janet*, la transcripción de voz en navegadores y algunas funcionalidades extra como mostrar más libros en una lista de libros.

Respecto a Rasa, he actualizado de la versión 1.7 a la 1.9, y junto a esto también he adaptado la funcionalidad para eliminar el servidor de Jarvis y poder ejecutar modelos de Rasa en Python de forma directa. También he manejado la obtención de entidades de los trackers, he podido añadir intenciones e historias nuevas, reescribir las antiguas, y he implementado la nueva petición de correos electrónicos.

Me he encargado plenamente de los clientes móviles. Esto incluye la implementación de las nuevas funcionalidades, el cambio de la interfaz gráfica, los modos de alto contraste y el arreglo de bugs. El despliege tanto a la App Store como a la Play Store también ha sido hecho por mi. Este trabajo incluye la creación de certificados para firmar las aplicaciones, la compilación de las versiones, la redacción del *changelog*, y la realización de capturas de pantalla representativas para cada dispositivo.

En el instalador, he parametrizado tanto el usuario como la ruta de instalación, y creado las actualizaciones de *Janet*, detectando las instalaciones existentes. Además de esto, he resuelto algunos problemas en el servidor como el problema del proxy inverso, además de ayudar con la resolución de bugs en todas las plataformas.

Respecto a la memoria, he escrito la introducción tanto en español como en inglés (1 y 2 respectivamente). En el apartado de "estado del arte" he redactado sobre los servicios y tecnologías de la biblioteca (3.1) y sobre los chatbots (3.2), buscando los diferentes tipos existentes y sus aplicaciones reales.

En la sección de tecnologías, he escrito sobre las tecnologías de desarrollo back end (4.2.1.1) y las tecnologías web de transcripción de voz (4.2.1.4). He podido contar también la problemática que supone implementar la transcripción en el entorno web.

En la sección del desarrollo he hablado sobre:

- El resumen de la arquitectura de la aplicación (5.2), dando una visión general sobre cómo funciona *Janet*.
- Las introducción narrando la actualización del bot de Rasa (5.3), junto a los cambios que esto implica.
- Los diversos tipos de mejoras que se han añadido a Rasa (5.3.1).
- El back end del cliente web (5.4).
- El desarrollo de las funcionalidades, rediseño gráfico, y los problemas de compatibilidad de las aplicaciones móviles (5.6).
- El instalador y actualizador de *Janet* (5.7).
- Los problemas con la arquitectura y las soluciones implementadas para solventarlos (5.8).

También he escrito el ejemplo de uso de la petición de un libro (6.1) y las peculiaridades que no son explicadas en dicho ejemplo (6.2). Por último, he redactado tanto las conclusiones como el trabajo futuro (7), además de ayudar con las correcciones generales en el trabajo.

9.2. Manuel María Guerrero Serrano

Mis aportaciones al proyecto han estado principalmente orientadas a lo relacionado con el back end y el procesamiento de lenguaje mediante Rasa, aunque en lo que respecta a la web, creé una plantilla inicial para mandar y recibir mensajes y comprobar conectividad con los servidores antiguos, y la cual terminó siendo la base de la web final.

En un inicio, rehice el instalador para solucionar los problemas que tenía el antiguo donde saltaban errores por dependencias y por incompatibilidad de SO. Añadí también el uso de un entorno virtual para que pudiéramos instalar distintas versiones en un mismo equipo.

Posteriormente, realicé algunos cambios necesarios en los servicios para systemd e hice la migración del sistema de logs para que pudieran ser accedidos mediante journalctl.

En lo que respecta a Rasa, realicé una ampliación de los casos de entrenamiento, tanto para el motor nlu con más formas de obtener una misma intención, como con más conversaciones de ejemplo, como en el dominio para disponer de un abanico más amplio de respuestas. Además, hice un port inicial a Rasa 1.7 en el que se actualizaba la manera de representar la información y que se terminó usando como base para la versión que hemos terminado utilizando, la 1.9. Por último, realicé la traducción parcial del modelo al inglés e implementé la funcionalidad para tener dos agentes, uno en inglés y el otro en español, y un sistema para reconocer el idioma de una petición y procesarla en el agente correspondiente.

Finalmente, llevé a cabo distintas tareas como pueden ser limpiar las bases de datos de títulos de libros y nombres de personas eliminando duplicados y añadiendo más, reubicar

archivos o renombrar variables junto a comentar el código para su mejor legibilidad, gestionar el enrutado para que las apps móviles pudieran acceder por el servidor web como intermediario o corregir distintos bugs que fueron surgiendo en el desarrollo del proyecto.

En cuanto a la memoria, en el capítulo 3, "Estado del Arte", he escrito sobre el estado en que encontramos la aplicación *Janet* al inicio del proyecto (3.3) y sobre el framework Rasa (3.4). En "Herramientas de desarrollo, tecnologías y metodología de trabajo" he descrito algunas herramientas que hemos utilizado (4.1).

En el capítulo 5, "Descripción del Trabajo" he escrito sobre:

- La visión general sobre los principales puntos a mejorar (5.1).
- La legibilidad del código y el sistema de logs (5.3.2).
- El proceso de traducción e implementación del agente en inglés (5.3.3).

Por último, también he escrito en las líneas de trabajo futuro (7) tanto en español como en inglés (8), además de haber revisado y corregido las partes de la memoria en inglés.

9.3. Mario Torres Cabañas

He participado en múltiples partes del desarrollo de este proyecto. Dentro del apartado Web me he encargado de la mayor parte de la sección del front end, tanto en el empleo de las clases de Bootstrap y los estilos CSS para definir el nuevo diseño de la aplicación, como para la implementación de los elementos interactivos con Javascript y Jquery. Esto incluye el desarrollo del modo de alto contraste, la lectura de voz por medio de TextToSpeech, y la implementación de los atributos ARIA, así como el ajuste de determinados elementos para que se ajustaran adecuadamente a este estándar. A su vez he desarrollado la venta de "Info" con sus diferentes ejemplos y enlaces.

Dentro de la sección de back end Web me he encargado del manejo de las direcciones, así como los redirección de los errores 404, y cambios en la implementación de Flask para asegurar el correcto funcionamiento de la página. En el apartado de la instalación en el servidor me encargué de la resolución de varios errores, entre ellos los que se produjeron en Flask, como el empleo de la variable SCRIPT_ROOT, así como de otros errores sucedidos con la web durante la instalación en el servidor.

En cuanto al apartado de RASA, me he encargado de realizar parte de la traducción del modelo al inglés. Así como la revisión y corrección de diversos errores tale como faltas de ortografía en los intents.

Respecto a la memoria, en el apartado de "Estado del Arte" he escrito acerca de los servicios de la biblioteca (3.1). Mientras que en la sección de "Herramientas de desarrollo, tecnologías y metodología de trabajo" he escrito la mayor parte de las herramientas empleadas (4.1) gran parte de sus subsecciones. Además de eso he redactado el apartado el punto acerca de las Tecnologías de desarrollo front end (4.2.1.2), el apartado acerca del funcionamiento de Bootstrap (4.2.1.3) y la sección acerca de la metodología de trabajo empleada durante el desarrollo (4.3).

En la sección del desarrollo he escrito sobre:

- La visión general de la aplicación (5.1).
- El back end del cliente web (5.4).
- El front end del cliente web (5.5).
- El instalador de Janet (5.7).
- Los problemas de arquitectura del servidor (5.8).

Además de todo lo anterior mencionado, me he encargado de la revisión de errores generales en el documento, de la redacción de parte de las conclusiones (7), y de la traducción al inglés de dicha sección (8).

Bibliografía

Allas Miguelsanzr, B. Be user friendly: los 10 principios de usabilidad web de jakob nielsen. https://profile.es/blog/los-10-principios-de-usabilidad-web-de-jakob-nielsen/, 2017. Último acceso: 23-06-2020.

Allen, J. F. *Natural Language Processing*, página 1218–1222. John Wiley and Sons Ltd., GBR, 2003. ISBN 0470864125.

Bradesko, L. y Mladenic, D. A survey of chabot systems through a loebner prize competition. En *Proceedings of Slovenian Language Technologies Society Eighth Conference of Language Technologies*. Artificial Intelligence laboratory, Jozef Stefan Institute, Ljubljana Slovenia, 2012.

Chaffer, J. y Swedberg, K. *Learning jQuery*. Packt Publishing, 2013.

Grinberg, M. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2014.

Kolodkin, E. The 10 most popular words users send to chatbots. https://chatbotsmagazine.com/10-most-popular-words-users-send-to-chatbots-98fc18a80b4a/, 2017. Último acceso: 23-06-2020.

Loureiro, M. A. y Varillas, J. L. M. Asistente virtual para servicios de la biblioteca de la ucm - janet. Facultad de Informática, Universidad Complutense de Madrid, España, 2019.

Ludo, M. *Bootstrap 4 Visual Learning Guide: a comprehensive example set for getting up to speed fast*. Code Blaze Books, 2019.

MDN. Accessible rich internet applications. https://developer.mozilla.org/es/docs/Web/Accessibility/ARIA, 2020. Último acceso: 23-06-2020.

Rasa. Rasa platform legacy documentation. https://legacy-docs.rasa.com/docs/platform/0.16.8/, 2018. Último acceso: 23-06-2020.

Rasa. Build contextual chatbots and ai assistants with rasa. https://rasa.com/docs/rasa/, 2020. Último acceso: 23-06-2020.

Guido van Rossum, N. C., Barry Warsaw. Pep 8 – style guide for python code. https://www.python.org/dev/peps/pep-0008/, 2001. Último acceso: 23-06-2020.

Shawar, B. A. y Atwell, E. Different measurements metrics to evaluate a chatbot system. En *Proceedings of the Workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technologies*, NAACL-HLT-Dialog '07, página 89–96. Association for Computational Linguistics, USA, 2007.

Valeur, F., Vigna, G., Kruegel, C. y Kirda, E. An anomaly-driven reverse proxy for web applications. En *Proceedings of the 2006 ACM symposium on Applied computing*, páginas 361–368. 2006.

Villar, A. L. 9 empresas usan chatbots: casos de éxito. `https://guelcom.net/9-empresas-usan-chatbots-casos-exito/`, 2018. Último acceso: 23-06-2020.

Wochinger, T. Profundizando en rasa nlu: Parte 1 — clasificación de intenciones. `https://planetachatbot.com/profundizando-en-rasa-nlu-clasificacion-intenciones-ea84c21881e`, 2019. Último acceso: 23-06-2020.